A High-Security Searchable Encryption Framework for Privacy-Critical Cloud Storage Services

Thang Hoang[*] Attila A. Yavuz[*] Jorge Guajardo[†]

Abstract

Searchable encryption has received a significant attention from the research community with various constructions being proposed, each achieving asymptotically optimal complexity for specific metrics (e.g., search, update). Despite their elegancy, the recent attacks and deployment efforts have shown that the optimal asymptotic complexity might not always imply practical performance, especially if the application demands a high privacy. Hence, there is a significant need for searchable encryption frameworks that capture the recent attacks with actual deployments on cloud infrastructures to assess the practicality under realistic settings.

In this article, we introduce a new Dynamic Searchable Symmetric Encryption (DSSE) framework called *Incidence Matrix (IM)-DSSE*, which achieves a high level of privacy, efficient search/update, and low client storage with actual deployments on real cloud settings. We harness an incidence matrix along with two hash tables to create an encrypted index, on which *both* search and update operations can be performed effectively with minimal information leakage. This simple set of data structures surprisingly offers a high level of DSSE security while at the same time achieving practical performance. Specifically, *IM-DSSE* achieves forward privacy, backward privacy and size-obliviousness properties simultaneously. We also create several DSSE variants, each offering different trade-offs (e.g., security, computation) that are suitable for different cloud applications and infrastructures. Our framework was fully-implemented and its performance was rigorously evaluated on a real cloud system (Amazon EC2). Our experimental results confirm that *IM-DSSE* is highly practical even when deployed on mobile phones with a large outsourced dataset. Finally, we have released our *IM-DSSE* framework as an open-source library for a wide development and adaptation.

Keywords— Privacy-enhancing technologies; private cloud services; dynamic searchable symmetric encryption

1 Introduction

The rise of cloud storage and computing services provides vast benefits to society and IT industry. One of the most important cloud services is data Storage-as-a-Service (SaaS), which can significantly reduce the cost of data management via continuous service, expertise and maintenance for resource-limited clients such as individuals or small/medium businesses. Despite its benefits, SaaS also brings significant security and privacy concerns to the user. That is, once a client outsources her data to the cloud, sensitive information (e.g., email) might be exploited by a malicious party (e.g., malware). Although standard encryption schemes (e.g., AES) can provide confidentiality, they also prevent the

[*]Oregon State University, {hoangmin, attila.yavuz}@oregonstate.edu

[†]Robert Bosch Research and Technology Center, Jorge.GuajardoMerchan@us.bosch.com

This research is kindly supported by Robert Bosch with an unrestricted gift. Part of this work was done while the first two authors were visiting the Bosch RTC—LLC, Pittsburgh, PA.

client from querying encrypted data from the cloud. This privacy versus data utilization dilemma may significantly degrade the benefits and usability of cloud-based systems. Therefore, it is vital to develop privacy-enhancing technologies that can address this problem while retaining the practicality of the underlying cloud service.

Searchable Symmetric Encryption (SSE) [9] enables a client to encrypt data in such a way that she can later perform keyword searches on it. These encrypted queries are performed via "search tokens" [27] over an encrypted index which represents the relationship between search token (keywords) and encrypted files. A prominent application of SSE is to enable privacy-preserving keyword search on the cloud (e.g., Amazon S3), where a data owner can outsource a collection of encrypted files and perform keyword searches on it without revealing the file and query contents [18]. Preliminary SSE schemes (e.g.,[26, 9]) only provide search-only functionality on static data (i.e., no dynamism), which strictly limits their applicability due to the lack of update capacity. Later, several Dynamic Searchable Symmetric Encryption (DSSE) schemes (e.g., [18, 5]) were proposed that permit the user to add and delete files after the system is set up. To the best of our knowledge, there is *no* single DSSE scheme that outperforms *all* the other alternatives in terms of *all* the aforementioned metrics: privacy (e.g., information leakage), performance (e.g., search, update delay), storage efficiency and functionality.

Research Gap and Objectives: Despite a number of DSSE schemes have been introduced in the literature, most of them only provide a theoretical asymptotic analysis[1] and in some cases, merely a prototype implementation. The lack of a rigorous actual experimental performance evaluation on real platforms poses a significant difficulty in assessing the application and practicality of proposed DSSE schemes, as the impacts of security vulnerability, hidden computation costs, multi-round communication delay and storage blowup might be overlooked. For instance, most efficient DSSE schemes (e.g., [5, 12]) are vulnerable to file-injection attacks, which have been showed to be easily conducted even by semi-honest adversary in practice, especially in the personal email scenario. Although several forward-secure DSSE schemes with an optimal asymptotic complexity have been proposed, they incur either very high delay due to public-key operations (e.g., [3]), or significant storage blowup at both client and server-side (e.g., [27]), and therefore, their ability to meet actual need of real systems in practice is still unclear.

There is a significant need for a DSSE scheme that can achieve a high level of security with a well-quantified information leakage, while maintaining a performance and functionality balance between the search and update operations. More importantly, it is critical that the performance of proposed DSSE should be experimentally evaluated in a realistic cloud environment with various parameter settings, rather than merely relying on asymptotic results. The investigation of alternative data structures and their optimized implementations on commodity hardware seem to be key factors towards achieving these objectives.

1.1 Our Contributions

In this article, towards filling the gaps between theory and practice in DSSE research community, we introduce IM-DSSE, a fully-implemented DSSE framework which favors desirable properties for realistic privacy-critical cloud systems including high security against practical attacks and low end-to-end delay. In this framework, we provide the full-fledged implementations of our preliminary DSSE scheme proposed in [31], as well as extended schemes, which are specially designed to fit with various application requirements and cloud data storage-as-a-service infrastructures in practice.

[1]One noticeable outlier is [5], which provides a standalone implementation.

Improvements over Preliminary Version: This article is the extended version of [31] which includes the following improvements: (i) We propose extended DSSE schemes which are more compatible with the cloud SaaS infrastructure and offer backward-privacy at the cost of bandwidth overhead. (ii) As a significant improvement over the preliminary version, we provide a comprehensive DSSE framework, where our preliminary DSSE scheme in [31] as well as all of its variants are fully implemented. We fully deployed our framework on Amazon EC2 cloud and provided a much more comprehensive performance analysis of each scheme with different hardware and network settings. (iii) Finally, we have released our framework for public use and improvement.

Desirable properties: IM-DSSE offers ideal features for privacy-critical cloud systems as follows.

- *Highly secure against File-Injection Attacks:* IM-DSSE offers *forward privacy* (see [27] or Section 4 for definition) which is an imperative security feature to mitigate the impact of practical file-injection attacks [3, 32]. Only few DSSE schemes offer this property (i.e., [27, 3]), some of which incur high client storage with costly update (e.g., [27]), or high delay due to public-key operations (e.g., [3]).

  Additionally, IM-DSSE offers *size-obliviousness* property, where it hides all size information involved with the encrypted index and update query including (i) update query size (i.e., number of keywords in the updated file); (iii) and the number of keyword-file pairs in the database. More importantly, one of the IM-DSSE variants achieves *backward privacy* defined in [27]. To the best of our knowledge, none of the state-of-the-art DSSE schemes offer all these security properties simultaneously.

- *Updates with Improved Features*: (i) IM-DSSE allows to *directly* update keywords of an existing file without invoking the file delete-then-add operation sequence. The update in IM-DSSE also leaks minimal information and it is type-oblivious, meaning that it does not leak timing information (i.e., all updates take the same amount of time) and whether the operation is add, delete, or update. (ii) The encrypted index of our schemes does not grow with update operations and, therefore, it does not require re-encryption due to frequent updates. This is more efficient than some alternatives (e.g., [27]) in which the encrypted index can grow linearly with the number of deletions.

- *Fully Parallelizable*: IM-DSSE supports parallelization for both update and search operations and, therefore, it takes full advantages of modern computing architecture to minimize the delay of cryptographic operations. Experiments on Amazon cloud indicates that the search latency of our framework is highly practical and mostly dominated by the network communication between the client and server (see Section 5).

- *Detailed experimental evaluation and open-source framework*: We deployed IM-DSSE in a realistic cloud environment (Amazon EC2) to assess the practicality of our framework. We experimented with different database sizes and investigated the impacts of network condition and storage unit on the overall performance. We also evaluated the performance of IM-DSSE on a resource-limited mobile client. We give a comprehensive cost breakdown analysis to highlight the main factors contributing the overall cost in all these settings. Finally, we released the implementation of our framework to public to provide opportunities for broad adaptation and testing (see Section 5).

2 IM-DSSE Framework

IM-DSSE framework comprises various DSSE schemes based on the incidence matrix data structure. In this section, we provide the detailed construction of the main scheme in IM-DSSE framework denoted as IM-DSSE$^{\text{main}}$, which is preliminarily presented in [31]. Several extension derived from IM-DSSE$^{\text{main}}$ scheme that IM-DSSE also fully supports will be described in the next section. We first start with notation, and then present typical data structures being used in IM-DSSE. We give the algorithmic details of IM-DSSE$^{\text{main}}$ scheme afterwards.

2.1 Notation and Data Structure

Notation. Operators $||$ and $|x|$ denote the concatenation and the bit length of variable x, respectively. $x \xleftarrow{\$} \mathcal{S}$ denotes variable x is randomly and uniformly selected from set $\mathcal{S}$. $(x_1, \ldots, x_n) \xleftarrow{\$} \mathcal{S}$ denotes $(x_1 \xleftarrow{\$} \mathcal{S}, \ldots, x_n \xleftarrow{\$} \mathcal{S})$. $|x|$ denotes the size of x. We denote $\{0,1\}^*$ as a set of binary strings of any finite length. $\lfloor x \rfloor$ and $\lceil x \rceil$ denote the floor and the ceiling of x, respectively. Given a matrix $\mathbf{I}$, $\mathbf{I}[i,j]$ denotes the cell indexing at row i and column j. $\mathbf{I}[*,j]$ and $\mathbf{I}[i,*]$ denote accessing column j and row i of matrix $\mathbf{I}$, respectively. $\mathbf{I}[*,a,\ldots,b]$ denotes accessing columns from a to b of matrix $\mathbf{I}$. $\mathbf{u}[i]$ denotes accessing the i'th component of vector $\mathbf{u}$.

We denote an IND-CPA encryption scheme as a triplet $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$: $k \leftarrow \mathcal{E}.\mathsf{Gen}(1^\kappa)$, where κ is a security parameter and k is a key; $c \leftarrow \mathcal{E}.\mathsf{Enc}_k(M, u)$ takes as input a secret key k, a counter u and a message M and returns a ciphertext c; $M \leftarrow \mathcal{E}.\mathsf{Dec}_k(c, u)$ takes as input a key k, a counter u and ciphertext c, and returns M if k and u were the key and the counter under which c was produced. The function G is a keyed Pseudo Random Function (PRF), denoted by $\tau \leftarrow G_k(x)$, which takes as input a secret key $k \xleftarrow{\$} \{0,1\}^\kappa$ and a string x, and returns a token/key r. We denote $H : \{0,1\}^{|x|} \longrightarrow \{0,1\}$ as a Random Oracle (RO) [1], which takes an input x and returns a bit.

IM-DSSE Data Structures. Our encrypted index is an incidence matrix $\mathbf{I}$, in which $\mathbf{I}[i,j].v \in \{0,1\}$ stores the (encrypted) relationship between keyword indexing at row i and file indexing at column j, and $\mathbf{I}[i,j].\mathsf{st} \in \{0,1\}$ stores a bit indicating the state of $\mathbf{I}[i,j].v$. Particularly, $\mathbf{I}[i,j].\mathsf{st}$ is set to 1 or 0 if $\mathbf{I}[i,j].v$ is accessed by update or search, respectively. For simplicity, we will often write $\mathbf{I}[i,j]$ to denote $\mathbf{I}[i,j].v$, and be explicit about the state bit as $\mathbf{I}[i,j].\mathsf{st}$.

The encrypted index $\mathbf{I}$ is augmented by two static hash tables T_w and T_f that associate a keyword and file to a unique row and a column, respectively. Specifically, T_f is a file static hash table whose key-value pair is $(s_{id_j}, \langle y_j, u_j \rangle)$, where $s_{id_j} \leftarrow G_{k_2}(id_j)$ for file with identifier id_j, column index $y_j \in \{1, \ldots, n\}$ equivalent to the index of s_{id_j} in T_f and u_j is a counter value. We denote access operations by $y_j \leftarrow T_f(s_{id_j})$ and $u_j \leftarrow T_f[y_j].\mathsf{ct}$. T_w is a keyword static hash table whose key-value pair is $\{s_{w_i}, \langle x_i, u_i \rangle\}$, where token $s_{w_i} \leftarrow G_{k_2}(w_i)$ for keyword w_i, row index $x_i \in \{1, \ldots, m\}$ equivalent to the index of s_{w_i} in T_w and u_i is a counter value. We denote access operations by $x_i \leftarrow T_w(s_{w_i})$ and $u_i \leftarrow T_w[x_i].\mathsf{ct}$. All counter values are incremental and initially set to 1. So, the client state information is in the form of T_w and T_f, that offers (on average) $\mathcal{O}(1)$ look-up time.

2.2 IM-DSSE$^{\text{main}}$ Algorithms

We present the detailed algorithmic construction of the main scheme in IM-DSSE framework in Scheme 1, which consists of nine algorithms with the following highlights.

• *Setup*: Given a file collection $\mathcal{F} = \{f_{id_1}, \ldots, f_{id_n}\}$ (with unique IDs $id_1, \ldots, id_n$) to be outsourced, the client extract all unique keywords $\mathcal{F}$, and constructs a matrix δ, which represents the relationship between keywords and outsourced files. Afterwards, the client invokes IM-DSSE.Gen algorithm to generate cryptographic keys which are used afterwards to encrypted δ and $\mathcal{F}$ (IM-DSSE.Enc algorithm) resulting in encrypted index γ and encrypted files $\mathcal{C} = \{c_{id_1}, \ldots, c_{id_n}\}$, respectively. Finally, the client sends γ and $\mathcal{C}$ to the cloud server along with the file hash table T_f.

• *Search*: To search keyword w, the client generates a token τ_w (IM-DSSE.SearchToken algorithm) containing a row index i and row key(s). Upon receiving τ_w, the server decrypts row i and determines column indexes j where $\mathbf{I}[i,j] = 1$, and returns the corresponding j-labeled ciphertexts to the client. Finally, the client invokes IM-DSSE.Dec algorithm on each ciphertext to obtain the search result.

• *Update*: To add (or delete) a file, the client generates a token containing column j and the encrypted file (if the file is added) via IM-DSSE.AddToken (IM-DSSE.DeleteToken resp.) algorithm. The server invokes IM-DSSE.Add ((IM-DSSE.Delete resp.)) algorithm to update column j of $\mathbf{I}$, and adds (deletes resp.) the file in $\mathcal{C}$.

Scheme 1 IM-DSSE$^{\text{main}}$ Scheme

$\mathcal{K} \leftarrow$ IM-DSSE.Gen(1^κ): Given security parameter κ, generate secret key $\mathcal{K}$

```
1: $k_1 \leftarrow \mathcal{E}.\mathsf{Gen}(1^\kappa)$ and $(k_2, k_3) \xleftarrow{\$} \{0,1\}^\kappa$
2: return $\mathcal{K}$, where $\mathcal{K} \leftarrow \{k_1, k_2, k_3\}$
```

$f \leftarrow$ IM-DSSE.Dec$_{\mathcal{K}}(c)$: Decrypt encrypted file c with key κ

```
1: $f \leftarrow \mathcal{E}.\mathsf{Dec}_{k_1}(c', y \| u)$ where $u \leftarrow T_f[y].\mathsf{ct}$, $(c', y) \leftarrow c$
2: return $f$
```

$(\gamma, \mathcal{C}) \leftarrow$ IM-DSSE.Enc$_{\mathcal{K}}(\delta, \mathcal{F})$: Given index δ and plaintext files $\mathcal{F}$, generate encrypted index γ and encrypted files $\mathcal{C}$

```
1: $T_w[i].\mathsf{ct} \leftarrow 1$, $T_f[j].\mathsf{ct} \leftarrow 1$, for $0 \le i \le m, 0 \le j \le n$
2: $\mathbf{I}[*,*].\mathsf{st} \leftarrow 0$ and $\delta[*,*] \leftarrow 0$
3: Extract $(w_1, \ldots, w_{m'})$ from $\mathcal{F} = \{f_{id_1}, \ldots, f_{id_{n'}}\}$
4: for $i = 1, \ldots, m'$ do
5:     $s_{w_i} \leftarrow G_{k_2}(w_i)$, $x_i \leftarrow T_w(s_{w_i})$
6:     for $j = 1, \ldots, n'$ do
7:         if $w_i$ appears in $f_{id_j}$ then
8:             $s_{id_j} \leftarrow G_{k_2}(id_j)$ and $y_j \leftarrow T_f(s_{id_j})$
9:             $\delta[x_i, y_j] \leftarrow 1$
10: for $i = 1, \ldots, m$ do
11:     $r_i \leftarrow G_{k_3}(i \| \overline{u}_i)$, where $\overline{u}_i \leftarrow T_w[i].\mathsf{ct}$
12:     for $j = 1, \ldots, n$ do
13:         $\mathbf{I}[i,j] \leftarrow \delta[i,j] \oplus H(r_i \| j \| u_j)$, where $u_j \leftarrow T_f[j].\mathsf{ct}$
14: for $j = 1, \ldots, n'$ do
15:     $c_j \leftarrow (c'_j, y_j)$, where $c'_j \leftarrow \mathcal{E}.\mathsf{Enc}_{k_1}(f_{id_j}, y_j \| u_{y_j})$
16: return $(\gamma, \mathcal{C})$, where $\gamma \leftarrow (\mathbf{I}, T_f)$ and $\mathcal{C} \leftarrow \{c_1, \ldots, c_{n'}\}$
```

$\tau_w \leftarrow$ IM-DSSE.SearchToken$(\mathcal{K}, w)$: Generate search token τ_w from keyword w and key $\mathcal{K}$

```
1: $s_w \leftarrow G_{k_2}(w)$, $i \leftarrow T_w(s_w)$
2: $\overline{u} \leftarrow T_w[i].\mathsf{ct}$, $r_i \leftarrow G_{k_3}(i \| \overline{u})$
3: if $\overline{u} = 1$ then
4:     $\tau_w \leftarrow (i, r_i)$
5: else
6:     $\overline{r}_i \leftarrow G_{k_3}(i \| \overline{u} - 1)$ and $\tau_w \leftarrow (i, r_i, \overline{r}_i)$
7: $T_w[i].\mathsf{ct} \leftarrow \overline{u} + 1$
8: return $\tau_w$
```

Scheme 1 IM-DSSE$^{\text{main}}$ Scheme (continued)

$(\mathcal{I}_w, \mathcal{C}_w) \leftarrow$ IM-DSSE.Search(τ_w, γ): Given search token τ_w and encrypted index γ, return sets of file identifiers $\mathcal{I}_w$ and encrypted files $\mathcal{C}_w \subseteq \mathcal{C}$ matching with τ_w

```
 1: for $j = 1, \ldots, n$ do
 2:     $u_j \leftarrow T_f[j].\mathrm{ct}$
 3:     if $(\tau_w = (i, r_i) \vee \mathbf{I}[i,j].\mathrm{st} = 1)$ then
 4:         $\mathbf{I}'[i,j] \leftarrow \mathbf{I}[i,j] \oplus H(r_i \| j \| u_j)$ and set $\mathbf{I}[i,j].\mathrm{st} \leftarrow 0$
 5:     else
 6:         $\mathbf{I}'[i,j] \leftarrow \mathbf{I}[i,j] \oplus H(\overline{r}_i \| j \| u_j)$ and set $\mathbf{I}[i,j] \leftarrow \mathbf{I}'[i,j] \oplus H(r_i \| j \| u_j)$
 7: $l \leftarrow 0$
 8: for each $j \in \{1, \ldots, n\}$ satisfying $\mathbf{I}'[i,j] = 1$ do
 9:     $l \leftarrow l + 1$ and $y_l \leftarrow j$
10: $\mathcal{I}_w \leftarrow \{y_1, \ldots, y_l\}$
11: $\gamma \leftarrow (\mathbf{I}, T_f)$, $\mathcal{C}_w \leftarrow \{(c_{y_1}, y_1), \ldots, (c_{y_l}, y_l)\}$
12: return $(\mathcal{I}_w, \mathcal{C}_w)$
```

$(\tau_f, c) \leftarrow$ IM-DSSE.AddToken$(\mathcal{K}, f_{id})$: Given key $\mathcal{K}$ and a file f_{id}, generate add token τ_f and ciphertext c of f_{id}

```
 1: $s_{id} \leftarrow G_{k_2}(id)$, $j \leftarrow T_f(s_{id})$, $T_f[j].\mathrm{ct} \leftarrow T_f[j].\mathrm{ct} + 1$ and $u_j \leftarrow T_f[j].\mathrm{ct}$
 2: for $i = 1, \ldots, m$ do
 3:     $r_i \leftarrow G_{k_3}(i \| \overline{u}_i)$, where $\overline{u}_i \leftarrow T_w[i].\mathrm{ct}$
 4: Extract $(w_1, \ldots, w_t)$ from $f_{id}$ and set $\overline{\mathbf{I}}[*, j] \leftarrow 0$
 5: for $i = 1, \ldots, t$ do
 6:     $s_{w_i} \leftarrow G_{k_2}(w_i)$, $x_i \leftarrow T_w(s_{w_i})$, $\overline{\mathbf{I}}[x_i, j] \leftarrow 1$
 7: for $i = 1, \ldots, m$ do
 8:     $\mathbf{I}'[i,j] \leftarrow \overline{\mathbf{I}}[i,j] \oplus H(r_i \| j \| u_j)$
 9: $c \leftarrow (c', j)$, where $c' \leftarrow \mathcal{E}.\mathrm{Enc}_{k_1}(f_{id}, j \| u_j)$
10: return $(\tau_f, c)$ where $\tau_f \leftarrow (\mathbf{I}', j)$
```

$(\gamma', \mathcal{C}') \leftarrow$ IM-DSSE.Add$(\gamma, \mathcal{C}, c, \tau_f)$: Add token τ_f and ciphertext c to encrypted index γ and set $\mathcal{C}$, resp.

```
 1: Set $\mathbf{I}[i,j] \leftarrow \mathbf{I}'[i,j]$ and $\mathbf{I}[i,j].\mathrm{st} \leftarrow 1$, for $1 \le i \le m$
 2: $T_f[j].\mathrm{ct} \leftarrow T_f[j].\mathrm{ct} + 1$
 3: return $(\gamma', \mathcal{C}')$, where $\gamma' \leftarrow (\mathbf{I}, T_f)$ and $\mathcal{C}' \leftarrow \mathcal{C} \cup \{(c, j)\}$
```

$\tau'_f \leftarrow$ IM-DSSE.DeleteToken$(\mathcal{K}, f)$: Given key $\mathcal{K}$ and deleted file f_{id}, generate deletion token τ'_f

```
 1: Execute steps (1-3) of AddToken Algorithm that produces $(j, u_j, \langle r_1, \ldots, r_m \rangle)$ and increases $T_f[j].\mathrm{ct}$ to 1
 2: for $i = 1, \ldots, m$ do
 3:     $\mathbf{I}'[i] \leftarrow H(r_i \| j | u_j)$
 4: return $\tau'_f$, where $\tau'_f \leftarrow (\mathbf{I}', j)$
```

$(\gamma', \mathcal{C}') \leftarrow$ IM-DSSE.Delete$(\gamma, \mathcal{C}, \tau'_f)$: Update token τ'_f in encrypted index γ' and delete a file from set $\mathcal{C}'$

```
 1: Set $\mathbf{I}[i,j] \leftarrow \mathbf{I}'[i,j]$ and $\mathbf{I}[i,j].\mathrm{st} \leftarrow 1$, for $1 \le i \le m$
 2: $T_f[j].\mathrm{ct} \leftarrow T_f[j].\mathrm{ct} + 1$
 3: return $(\gamma', \mathcal{C}')$, where $\gamma' \leftarrow (\mathbf{I}, T_f)$, $\mathcal{C}' \leftarrow \mathcal{C} \setminus \{(c, j)\}$
```

Keyword update for existing files. Some existing schemes (e.g., [24]) only permit adding or deleting a file, but do not permit updating keywords in an existing file *directly*. It is easy to achieve this in our scheme as follows: Assume the client wants to update file f_{id} by adding (or removing) some keywords, she will prepare a new column $\mathbf{I}'[i,j] \longleftarrow b_i$ for $1 \leq i \leq m$, where $b_i = 1$ if w_i is added and $b_i = 0$ if otherwise and $j \longleftarrow T_f(s_{id})$ with $s_{id} \longleftarrow G_{k_2}(id)$ as in IM-DSSE.AddToken algorithm (steps 4-6). The rest of the algorithm remains the same.

Analytical analysis. For keyword search, IM-DSSE$^{\text{main}}$ incurs n invocations of hash function H and n XOR operations. Despites the fact that IM-DSSE$^{\text{main}}$ has linear search complexity which is asymptotically less efficient than other DSSE schemes (e.g., [5, 3]), we show in the experiment that, this impact is *insignificant* in practice for personal cloud usage with moderate database size where all optimizations are taken into account. Specifically, since IM-DSSE$^{\text{main}}$ is fully parallelizable, the search and update computation times can be reduced to n/p and m/p, respectively, where p is the number of processors in the system. Therefore, cryptographic operations in IM-DSSE$^{\text{main}}$ only contribute a small portion to the overall end-to-end search delay which is dominated by the network communication latency between client and server. Moreover, notice that all sub-linear DSSE schemes [27, 5] are less secure and sometimes incur more costly updates than IM-DSSE$^{\text{main}}$. For file update operation, IM-DSSE$^{\text{main}}$ incurs m invocations of H and m XOR operations along with m bits of transmission.

Regarding to storage overhead, IM-DSSE$^{\text{main}}$ costs $\Big(2m \cdot n + n \cdot \big(\kappa + |u|\big)\Big)$ bits at the server for encrypted index $\mathbf{I}$ and file hash table T_f. At the client side, IM-DSSE$^{\text{main}}$ requires $(n+m)\big(\kappa + |u|\big) + 3\kappa$ bits for two hash tables T_w, T_f and secret key $\boldsymbol{\mathcal{K}}$.

3 Extended IM-DSSE Schemes

We now present efficient extended schemes derived from IM-DSSE$^{\text{main}}$ scheme in Section 2 that our IM-DSSE framework also supports.

3.1 IM-DSSE$^{\text{I}}$: Minimized search latency

In IM-DSSE$^{\text{main}}$, we encrypt each cell of $\mathbf{I}$ with a unique key-counter pair, which requires n invocations of H during keyword search. This might not be ideal for some applications that require extremely prompt search delay. Hence, we introduce an extended scheme called IM-DSSE$^{\text{I}}$, which aims at achieving a very low search latency with the cost of increasing update delay. Specifically, instead of encrypting the index bit-by-bit as in IM-DSSE$^{\text{main}}$ scheme, IM-DSSE$^{\text{I}}$ leverages b-bit block cipher encryption to encrypt b successive cells with the same key-counter pair. This is achieved by interpreting columns of $\mathbf{I}$ as $D = \lceil \frac{n}{b} \rceil$ blocks, each being IND-CPA encrypted using counter (CTR) mode with block cipher size b. The counter will be stored via a block counter array (denoted $\mathbf{u}$) instead of $T_f[\cdot].u$ as in the main scheme. The update state is maintained for each block rather than each cell of $\mathbf{I}[i,j]$. Hence, $\mathbf{I}$ is decomposed into two matrices with different sizes: $\mathbf{I}.v \in \{0,1\}^{m \times n}$ and $\mathbf{I}.\text{st} \in \{0,1\}^{m \times D}$.

IM-DSSE$^{\text{I}}$ requires some straightforward algorithmic modifications from the main scheme. Specifically, we substitute encryption and decryption using $H(r_i \| j \| u_j)$ with $\mathcal{E}.\text{Enc}_{r_i}(\cdot, l \| u'_l)$ and $\mathcal{E}.\text{Dec}_{r_i}(\cdot, l \| u'_l)$, respectively, where u_l is a block counter stored in $\mathbf{u}$. Since $\mathbf{I}$ is encrypted by blocks, the client needs to retrieve a whole block and the states first before being able to update a column residing in the block during file update. Therefore, the reduction of search cost increases the cost of communication overhead for the update as a trade-off. We present modified algorithms for file addition in Scheme 2. The modifications for file deletion follow the same principle. The Gen and Dec algorithms of IM-DSSE$^{\text{I}}$ are identical to those of the main scheme.

Scheme 2 IM-DSSE$^{\text{I}}$ Scheme

$(\tau_f, c) \leftarrow \mathsf{AddToken}(\mathcal{K}, f_{id})$: Given key $\mathcal{K}$ and a file f_{id}, generate addition token τ_f and ciphertext c of f_{id}

```
1: $s_{id} \leftarrow G_{k_2}(id)$, $j \leftarrow T_f(s_{id})$, $l \leftarrow \left\lfloor \frac{j-1}{b} \right\rfloor$, $u_l \leftarrow \mathbf{u}[l]$
2: $a \leftarrow (l \cdot b) + 1$, $a' \leftarrow b \cdot (l + 1)$
3: Extract $(w_1, \dots, w_t)$ from $f_{id}$
4: for $i = 1, \dots, t$ do
5:     $s_{w_i} \leftarrow G_{k_2}(w_i)$, $x_i \leftarrow T_w(s_{w_i})$
6: Get from server $(\mathbf{I}[*, a \dots a'])$ and $\mathbf{I}[*, l].\mathrm{st}$
7: for $i = 1, \dots, m$ do
8:     $\overline{u}_i \leftarrow T_w[i].\mathrm{ct}$
9:     if $(\overline{u}_i > 1 \wedge \mathbf{I}[i, l].\mathrm{st} = 0)$ then
10:        $\overline{u}_i \leftarrow \overline{u}_i - 1$
11:    $r_i \leftarrow G_{k_3}(i \| \overline{u}_i)^{\dagger}$
12:    $\mathbf{I}'[i, a \dots, a'] \leftarrow \mathcal{E}.\mathsf{Dec}_{r_i}(\mathbf{I}[i, a \dots a'], l \| u_l)$
13: $\mathbf{I}'[i, j] \leftarrow 0$ for $1 \le i \le m$ and $\mathbf{I}'[x_i, j] \leftarrow 1$ for $1 \le i \le t$
14: $\mathbf{u}[l] \leftarrow \mathbf{u}[l] + 1$, $u_l \leftarrow \mathbf{u}[l]$
15: for $i = 1, \dots, m$ do
16:    if $(\overline{u}_i > 1 \wedge \mathbf{I}[i, l].\mathrm{st} = 0)$ then
17:        $r_i \leftarrow G_{k_3}(i \| \overline{u}_i + 1)$
18:    $\bar{\mathbf{I}}[i, a \dots a'] \leftarrow \mathcal{E}.\mathsf{Enc}_{r_i}(\mathbf{I}'[i, a \dots a'], l \| u_l)$
19: $T_f[j].\mathrm{ct} \leftarrow T_f[j].\mathrm{ct} + 1$ and $u'_j \leftarrow T_f[j].\mathrm{ct}$
20: $c \leftarrow (c', j)$ where $c' \leftarrow \mathcal{E}.\mathsf{Enc}_{k_1}(f_{id}, j \| u'_j)$
21: return $(\tau_f, c)$ where $\tau_f \leftarrow (\bar{\mathbf{I}}, j)$
```

$(\gamma', \mathcal{C}') \leftarrow \mathsf{Add}(\gamma, \mathcal{C}, c, \tau_f)$: Add token τ_f and ciphertext $\mathcal{C}$ to encrypted index γ and ciphertext set $\mathcal{C}$, resp.

```
1: $l \leftarrow \left\lfloor \frac{j-1}{b} \right\rfloor$, $a \leftarrow (l \cdot b) + 1$, $a' \leftarrow b(l + 1)$
2: $\mathbf{I}[i, j'] \leftarrow \bar{\mathbf{I}}[i, j']$, for $1 \le i \le m$ and $a \le j' \le a'$
3: $\mathbf{u}[l] \leftarrow \mathbf{u}[l] + 1$ and $\mathbf{I}[*, l].\mathrm{st} \leftarrow 1$
4: return $(\gamma', \mathcal{C}')$, where $\gamma' \leftarrow (\mathbf{I}, T_f)$ and $\mathcal{C}' \leftarrow \mathcal{C} \cup \{c\}$
```

† G should generate a suitable key for $\mathcal{E}$ (e.g., 128-bit key for AES-CTR)

Analytical analysis. For keyword search, IM-DSSE$^{\text{I}}$ requires n/b invocations of $\mathcal{E}$, which is theoretically b times faster than the main scheme. Given the CTR mode, the search time can be reduced to $n/(b \cdot p)$, where p is the number of processors in the system. For update, IM-DSSE$^{\text{I}}$ requires transmission of $(2b + 1) \cdot m$ bits along with decryption and encryption operations at the client side, compared with m non-interactive transmission and encryption-only in the main scheme. Thus, IM-DSSE$^{\text{I}}$ offers a trade-off where the search speed is increased by a factor of b (e.g., $b = 128$) with the cost of transmitting $(2b + 1) \cdot m$ bits in update operation. IM-DSSE$^{\text{I}}$ reduces the server storage overhead to $\left(\frac{n \cdot |u| + m \cdot n \cdot (b+1)}{b} \right)$ bits, while the client storage remains the same as in IM-DSSE$^{\text{main}}$.

3.2 IM-DSSE$^{\text{II}}$: Achieving cloud SaaS infrastructure with backward privacy

All DSSE schemes introduced so far require the server to perform some computation (i.e., encryption/decryption) during keyword search, which might not be fully compatible with typical cloud systems (e.g., Dropbox, Google Drive, Amazon S3) that generally only offers storage-only services. Hence,

Scheme 3 IM-DSSE$^{\text{II}}$ Scheme

$(\mathcal{I}_w, \mathcal{C}_w) \leftarrow \text{Search}(\mathcal{K}, w)$: Given keyword w and key $\mathcal{K}$, return sets of file identifiers $\mathcal{I}_w$ and encrypted files $\mathcal{C}_w \subseteq \mathcal{C}$ matching with w

```
1: $s_{w_i} \leftarrow G_{k_2}(w)$, $i \leftarrow T_w(s_{w_i})$, $r_i \leftarrow G_{k_3}(i)$, $l \leftarrow 0$
2: Fetch the $i$'th row data $\mathbf{I}[i, *]$ from server
3: for $j = 1, \dots, n$ do
4:     $u_j \leftarrow T_f[j]$.ct
5:     $\mathbf{I}'[i, j] \leftarrow \mathbf{I}[i, j] \oplus H(r_i \| j \| u_j)$
6: for each $j \in \{1, \dots, n\}$ satisfying $\mathbf{I}'[i, j] = 1$ do
7:     $l \leftarrow l + 1$ and $y_l \leftarrow j$
8: $\mathcal{I}_w \leftarrow \{y_1, \dots, y_l\}$
9: Send $\mathcal{I}_w$ to server and receive $\mathcal{C}_w = \{(c_{y_1}, y_1), \dots, (c_{y_l}, y_l)\}$
10: $f_i \leftarrow \text{Dec}_{\mathcal{K}}(c_{y_i})$ for $1 \le i \le l$
11: return $(\mathcal{I}_w, \mathcal{F}_w)$, where $\mathcal{F}_w \leftarrow \{f_1, \dots, f_l\}$
```

we propose another extended scheme derived from IM-DSSE$^{\text{main}}$ scheme called IM-DSSE$^{\text{II}}$, where all computations during keyword search are performed at the client side while the server does nothing rather than serving as a storage service. This simple trick makes IM-DSSE$^{\text{II}}$ not only compatible with the current infrastructure of SaaS clouds, but also more importantly, achieve the backward-privacy property. This is because the server is no longer able to decrypt any part of encrypted index to keep track of historical update operations. Additionally, IM-DSSE$^{\text{II}}$ also reduces the storage overhead at both client and server sides by eliminating the need of state matrix and keywords counters, which is used in IM-DSSE$^{\text{main}}$ and IM-DSSE$^{\text{I}}$ schemes to perform correct decryption during keyword search and forward-privacy during update. The detail is as follows.

To search a keyword w, the client sends to the server the w's row index i and receives the corresponding row. The client decrypts row i, obtains the column indexes j, where $\mathbf{I}[i, j] = 1$. The client then fetches and decrypts encrypted files indexed at j to obtain the search result. We present the keyword search of IM-DSSE$^{\text{II}}$ in Scheme 3, which is a protocol combined from IM-DSSE.SearchToken and IM-DSSE.Search algorithms in IM-DSSE$^{\text{main}}$ scheme. Since everything is computed by the client, it is not required to derive new keys for forward-privacy and therefore, state matrix $\mathbf{I}[*, *]$.st as well as file hash table T_f at the server and keyword counters T_w.ct at the client are not needed in IM-DSSE$^{\text{II}}$. Therefore, the modifications of IM-DSSE.Enc, IM-DSSE.Add, IM-DSSE.AddToken, IM-DSSE.Delete, IM-DSSE.DeleteToken algorithms are straightforward by (i) substituting row key generation $r_i \leftarrow G_{k_3}(i, \overline{u}_i)$ by $r_i \leftarrow G_{k_3}(i)$, (ii) omitting all keyword counters $\overline{u}_i$, block states $\mathbf{I}[*, *]$.st, T_f at the server (e.g., step 2, IM-DSSE.Add algorithm) and all operations involved.

Analytical analysis. The computation cost of IM-DSSE$^{\text{II}}$ is identical to the main scheme (i.e., n and m invocations of H for search and update resp.), except that the decryption is performed at the client, instead of the server during keyword search. However, IM-DSSE$^{\text{II}}$ requires m bits of transmission and a two-round communication. IM-DSSE$^{\text{II}}$ reduces the client and sever storage costs to $n\big(\kappa + |u|\big) + m \cdot \kappa + 3\kappa$ and $m \cdot n$ bits, respectively.

3.3 IM-DSSE$^{\text{I+II}}$: Low search latency, backward-privacy and compatible with cloud SaaS infrastructure

Our IM-DSSE also supports IM-DSSE$^{\text{I+II}}$, an extended DSSE scheme which is the combination between IM-DSSE$^{\text{I}}$ and IM-DSSE$^{\text{II}}$ schemes. Specifically in IM-DSSE$^{\text{I+II}}$, the incidence matrix $\mathbf{I}$ is encrypted with b-bit block cipher encryption, and the decryption is performed by the client during search. Since

$\mathsf{IM\text{-}DSSE}^{\mathsf{I+II}}$ inherits all properties of $\mathsf{IM\text{-}DSSE}^{\mathsf{I}}$ and $\mathsf{IM\text{-}DSSE}^{\mathsf{II}}$ schemes, $\mathsf{IM\text{-}DSSE}^{\mathsf{I+II}}$ is highly desirable for cloud SaaS infrastructure that requires a very low search latency and backward-privacy with the costs of more delayed update and an extra communication round during search.

4 Security Analysis

In this section, we analyze the security and update privacy of all DSSE schemes provided in our IM-DSSE framework.

4.1 Security Model

Most known efficient SSE schemes (e.g., [27, 5, 24]) reveal the *search and file-access patterns* defined as follows:

- Given search query w at time t, *the search pattern* $\mathcal{P}(\delta, \text{Query}, t)$ is a binary vector of length t with a 1 at location i if the search time $i \le t$ was for w, and 0 otherwise. The search pattern indicates whether the same keyword has been searched in the past or not.

- Given search query w_i at time t, *the file-access pattern* $\Delta(\delta, \mathcal{F}, w, t)$ is identifiers $\mathcal{I}_w$ of files having w_i.

We consider the following leakage functions, in the line of [16] that captures dynamic file addition/deletion in its security model as we do, but we leak much less information compared to [16].

Definition 1. *We define leakage functions* $(\mathcal{L}_1, \mathcal{L}_2)$ *as follows:*

1. $(m, n, \mathcal{I}, \langle |f_{id_1}|, \ldots, |f_{id_n}| \rangle) \longleftarrow \mathcal{L}_1(\delta, \mathcal{F})$*: Given the index* δ *and the set of files* $\mathcal{F}$ *(including their identifiers),* $\mathcal{L}_1$ *outputs the maximum number of keywords* m*, the maximum number of files* n*, the identifiers* $\mathcal{I} = \{id_1, \ldots, id_n\}$ *of* $\mathcal{F}$ *and the size of file* $|f_{id_j}|$ *for* $1 \le j \le n$ *(which also implies the size of its corresponding ciphertext* $|c_{id_j}|$*).*

2. $(\mathcal{P}(\delta, Query, t), \Delta(\delta, \mathcal{F}, w, t)) \longleftarrow \mathcal{L}_2(\delta, \mathcal{F}, w, t)$*: Given the index* δ*, the set of files* $\mathcal{F}$ *and a keyword* w *for a search operation at time* t*, it outputs the search pattern* $\mathcal{P}$ *and file-access pattern* Δ*.*

Definition 2 (IND-CKA2 Security [9, 18])**.** *Let* $\mathcal{A}$ *be a stateful adversary and* $\mathcal{S}$ *be a stateful simulator. Consider the following probabilistic experiments:*

Real$_{\mathcal{A}}(\kappa)$*: The challenger executes* $\mathcal{K} \leftarrow \mathsf{Gen}(1^\kappa)$*.* $\mathcal{A}$ *produces* $(\delta, \mathcal{F})$ *and receives* $(\gamma, \mathcal{C}) \leftarrow \mathsf{Enc}_{\mathcal{K}}(\delta, \mathcal{F})$ *from the challenger.* $\mathcal{A}$ *makes a polynomial number of adaptive queries* $\mathsf{Query} \in (w, f_{id}, f_{id'})$ *to the challenger. If* $\mathsf{Query} = w$ *is a keyword search query then* $\mathcal{A}$ *receives a search token* $\tau_w \longleftarrow \mathsf{SearchToken}(\mathcal{K}, w)$ *from the challenger. If* $\mathsf{Query} = f_{id}$ *is a file addition query then* $\mathcal{A}$ *receives an addition token* $(\tau_f, c) \longleftarrow \mathsf{AddToken}(\mathcal{K}, f_{id})$ *from the challenger. If* $\mathsf{Query} = f_{id'}$ *is a file deletion query then* $\mathcal{A}$ *receives a deletion token* $\tau'_f \longleftarrow \mathsf{DeleteToken}(\mathcal{K}, f_{id'})$ *from the challenger. Eventually,* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

Ideal$_{\mathcal{A},\mathcal{S}}(\kappa)$*:* $\mathcal{A}$ *produces* $(\delta, \mathcal{F})$*. Given* $\mathcal{L}_1(\delta, \mathcal{F})$*,* $\mathcal{S}$ *generates and sends* $(\gamma, \mathcal{C})$ *to* $\mathcal{A}$*.* $\mathcal{A}$ *makes a polynomial number of adaptive queries* $\mathsf{Query} \in (w, f_{id}, f_{id'})$ *to* $\mathcal{S}$*. For each query,* $\mathcal{S}$ *is given* $\mathcal{L}_2(\delta, \mathcal{F}, w, t)$*. If* $\mathsf{Query} = w$ *then* $\mathcal{S}$ *returns a simulated search token* τ_w*. If* $\mathsf{Query} = f_{id}$ *or* $\mathsf{Query} = f_{id'}$*,* $\mathcal{S}$ *returns a simulated addition token* τ_f *or deletion token* τ'_f *,respectively. Eventually,* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

A DSSE *is said to be* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure against adaptive chosen-keyword attacks (CKA2-security) if for all* PPT *adversaries* $\mathcal{A}$*, there exists a* PPT *simulator* $\mathcal{S}$ *such that*

$$|\Pr[\mathbf{Real}_{\mathcal{A}}(\kappa) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa) = 1]| \quad \leq \quad \mathsf{neg}(\kappa)$$

Remark 1. *In Definition 2, we adopt the notion of* dynamic CKA2-security *from [16], which captures the file addition and deletion operations by simulating tokens* τ_f *and* τ'_f*, respectively.*

The security of IM-DSSE can be stated as follows.

Theorem 1. *If* Enc *is IND-CPA secure, G is PRF and H is a Random Oracle (RO) then IM-DSSE is* $(\mathcal{L}_1, \mathcal{L}_2)$*-secure in ROM by Definition 2 (CKA-2 security with update capacity).*

Proof. We present the detailed IND-CKA2 security proof for IM-DSSE$^{\mathsf{main}}$ scheme in Section 2. The proof for extended schemes in Section 3 can be easily derived from this proof and therefore, we will not repeat it.

To begin with, we construct a simulator $\mathcal{S}$ that interacts with an adversary $\mathcal{A}$ in an execution of an $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa)$ experiment as described in Definition 2. In this experiment, $\mathcal{S}$ maintains lists $\mathcal{LR}$, $\mathcal{LK}$ and $\mathcal{LH}$ to keep track of query results, states and history information, respectively. Initially, all lists are set to empty. $\mathcal{LR}$ is a list of key-value pairs and is used to keep track of $RO(\cdot)$ queries. We denote $\mathsf{value} \leftarrow \mathcal{LR}(\mathsf{key})$ and $\perp \leftarrow \mathcal{LR}(\mathsf{key})$ if key does not exist in $\mathcal{LR}$. $\mathcal{LK}$ is to keep track of random values generated during the simulation and it follows the same as $\mathcal{LR}$. $\mathcal{LH}$ is to keep track of search and update queries, $\mathcal{S}$'s replies to those queries and their leakage output from $(\mathcal{L}_1, \mathcal{L}_2)$. $\mathcal{S}$ executes the simulation as follows:

I. Handle $RO(\cdot)$ Queries: $b \leftarrow RO(x)$ takes an input x and returns a bit b as output. Given x, if $\perp = \mathcal{LR}(x)$ set $b \xleftarrow{\$} \{0,1\}$, insert (x, b) into $\mathcal{LR}$ and return b as the output. Else, return $b \leftarrow \mathcal{LR}(x)$ as the output.

II. Simulate $(\gamma, \mathcal{C})$: Given $(m, n, \langle id_1, \ldots, id_{n'} \rangle, \langle |c_1|, \ldots, |c_{n'}| \rangle) \leftarrow \mathcal{L}_1(\delta, \mathcal{F})$, $\mathcal{S}$simulates $(\gamma, \mathcal{C})$ as follows:

1. $(s_{id_j}, k) \xleftarrow{\$} \{0,1\}^{\kappa}$, $y_j \leftarrow T_f(s_{id_j})$, insert (id_j, s_{id_j}, y_j) into $\mathcal{LH}$ and $c_{y_j} \leftarrow \mathcal{E}.\mathsf{Enc}_k(\{0\}^{|c_{id_j}|})$ for $1 \leq j \leq n'$.
2. For $j = 1, \ldots, n$ and $i = 1, \ldots, m$
   (a) $T_w[i].\mathrm{ct} \leftarrow 1$ and $T_f[j].\mathrm{ct} \leftarrow 1$.
   (b) $z_{i,j} \xleftarrow{\$} \{0,1\}^{\kappa}$, $\mathbf{I}[i,j] \leftarrow RO(z_{i,j})$ and $\mathbf{I}[i,j].\mathrm{st} \leftarrow 0$.
3. Output $(\gamma, \mathcal{C})$, where $\gamma \leftarrow (\mathbf{I}, T_f)$ and $\mathcal{C} \leftarrow \{\langle c_i, y_i \rangle\}_{i=1}^{n'}$

Correctness and Indistinguishability of the Simulation: $\mathcal{C}$ has the correct size and distribution, since $\mathcal{L}_1$ leaks $\langle |c_{id_1}|, \ldots, |c_{id_{n'}}| \rangle$ and Enc is a IND-CPA secure scheme, respectively. $\mathbf{I}$ and T_f have the correct size since $\mathcal{L}_1$ leaks (m, n). Each $\mathbf{I}[i,j]$ for $1 \leq j \leq n$ and $1 \leq i \leq m$ has random uniform distribution, since $RO(\cdot)$ is invoked with random value $z_{i,j}$. T_f has the correct distribution, since each s_{id_j} has random uniform distribution, for $1 \leq j \leq n'$. Hence, $\mathcal{A}$ does not abort due to $\mathcal{A}$'s simulation of $(\gamma, \mathcal{C})$. The probability that $\mathcal{A}$ queries $RO(\cdot)$ on any $z_{i,j}$ before $\mathcal{S}$ provides $\mathbf{I}$ to $\mathcal{A}$is negligible (i.e., $\frac{1}{2^{\kappa}}$). Hence, $\mathcal{S}$ also does not abort.

III. Simulate τ_w: Simulator $\mathcal{S}$ receives a search query for an arbitrary keyword w on time t. $\mathcal{S}$is given $\big(\mathcal{P}(\delta, \mathrm{Query}, t), \Delta(\delta, \mathcal{F}, w, t)\big) \leftarrow \mathcal{L}_2(\delta, \mathcal{F}, w, t)$. $\mathcal{S}$ adds these to $\mathcal{LH}$. $\mathcal{S}$then simulates τ_w and updates lists $(\mathcal{LR}, \mathcal{LK})$ as follows:

1. If w is in $\mathcal{LH}$, then fetch s_w. Else, $s_w \overset{\$}{\longleftarrow} \{0,1\}^{\kappa}$, $i \longleftarrow T_w(s_{w_i})$, $\overline{u}_i \longleftarrow T_w[i].\mathrm{ct}$, insert $(w, \mathcal{L}_1(\delta, \mathcal{F}), s_w)$ into $\mathcal{LH}$.
2. If $\perp = \mathcal{LK}(i\|\overline{u}_i)$, then $r_i \overset{\$}{\longleftarrow} \{0,1\}^{\kappa}$ and insert $(r_i, i, \overline{u}_i)$ into $\mathcal{LK}$. Else, $r_i \longleftarrow \mathcal{LK}(i\|\overline{u}_i)$.
3. If $\overline{u}_i > 1$, then $\overline{r}_i \longleftarrow \mathcal{LK}(i\|\overline{u}_i - 1)$, $\tau_w \longleftarrow (i, r_i, \overline{r}_i)$. Else, $\tau_w \longleftarrow (i, r_i)$.
4. $T_w[i].\mathrm{ct} \longleftarrow \overline{u}_i + 1$.
5. Given $\mathcal{L}_2(\delta, \mathcal{F}, w, t)$, $\mathcal{S}$ knows identifiers $\mathcal{I}_w = \{y_1, \ldots, y_l\}$. Set $\mathbf{I}'[i,y] \longleftarrow 1$ for each $y \in \mathcal{I}_w$ and rest of the elements as $\mathbf{I}'[i,j] \longleftarrow 0$ for each $j \in \left\{\{1,\ldots,n\} \setminus \mathcal{I}_w\right\}$.
6. If $((\tau_w = (i, r_i) \vee \mathbf{I}[i,j].\mathrm{st}) = 1)$, then $\mathbf{V}[i,j] \longleftarrow \mathbf{I}[i,j]' \oplus \mathbf{I}[i,j]$ and insert tuple $(r_i\|j\|u_j, \mathbf{V}[i,j])$ into $\mathcal{LR}$, where $u_j \longleftarrow T_f[j].\mathrm{ct}$ for $1 \leq j \leq n$.
7. $\mathbf{I}[i,j].\mathrm{st} \longleftarrow 0$ for $1 \leq j \leq n$.
8. $\mathbf{I}[i,j] \longleftarrow \mathbf{I}'[i,j] \oplus RO(r_i\|j\|u_j)$, where $u_j \longleftarrow T_f[j].\mathrm{ct}$ for $1 \leq j \leq n$.
9. Output τ_w and insert (w, τ_w) into $\mathcal{LH}$.

Correctness and Indistinguishability of the Simulation: Given any $\Delta(\delta, \mathcal{F}, w, t)$, $\mathcal{S}$ simulates the output of $RO(\cdot)$ such that τ_w always produces the correct search result for $\mathcal{I}_w \longleftarrow \mathsf{Search}\tau_w, \gamma$. $\mathcal{S}$ needs to simulate the output of $RO(\cdot)$ for two conditions (as in *III-step 6*): (i) The first search of w (i.e., $\tau_w \overset{?}{=} (i, r_i)$), since $\mathcal{S}$ did not know δ during the simulation of $(\gamma, \mathcal{C})$. (ii) If any file f_{id} containing w has been updated after the last search on w (i.e., $\mathbf{I}[i,j].\mathrm{st} \overset{?}{=} 1$), since $\mathcal{S}$ does not know the update content. $\mathcal{S}$ sets the output of $RO(\cdot)$ for those cases by inserting tuple $(r_i\|j\|u_j, \mathbf{V}[i,j])$ into $\mathcal{LR}$ (as in *III-step 6).* In other cases, $\mathcal{S}$ just invokes $RO(\cdot)$ with $(r_i\|j\|u_j)$, which consistently returns the previously inserted bit from $\mathcal{LR}$ (as in *III-step 8*).

During the first search on w, each $RO(\cdot)$ outputs $\mathbf{V}[i,j] = RO(r_i\|j|u_j)$ that has the correct distribution, since $\mathbf{I}[i,*]$ of γ has random uniform distribution (see *II-Correctness and Indistinguishability* argument). Let $\mathcal{J} = \{j_1, \ldots, j_l\}$ be the set of indexes of files containing w, which are updated after the last search on w. If w is searched again after being updated, then each $RO(\cdot)$'s output $\mathbf{V}[i,j] = RO(r_i\|j|u_j)$ has the correct distribution, since $\tau_f \longleftarrow (\mathbf{I}', j)$ for indexes $j \in \mathcal{J}$ has random uniform distribution (see *IV-Correctness and Indistinguishability* argument). Given that $\mathcal{S}$'s τ_w always produces correct $\mathcal{I}_w$ for given $\Delta(\delta, \mathcal{F}, w, t)$, and relevant values and $RO(\cdot)$ outputs have the correct distribution, $\mathcal{A}$ does not abort during the simulation due to $\mathcal{S}$'s search token. The probability that $\mathcal{A}$ queries $RO(\cdot)$ on any $(r_i\|j|u_j)$ before querying $\mathcal{S}$ on τ_w is negligible (i.e., $\frac{1}{2^{\kappa}}$) and, therefore, $\mathcal{S}$ does not abort due to $\mathcal{A}$'s search query.

IV. Simulate (τ_f, τ_f'): $\mathcal{S}$ receives an update request Query = ($\langle\mathsf{Add}, |c|\rangle$, Delete) for an arbitrary file having id at time t. $\mathcal{S}$simulates update tokens (τ_f, τ_f') as follows:

1. If id is in $\mathcal{LH}$, then fetch (id, s_{id}, j). Else set $s_{id} \overset{\$}{\longleftarrow} \{0,1\}^{\kappa}$, $j \longleftarrow T_f(s_{id})$ and insert (id, s_{id_j}, j) into $\mathcal{LH}$.
2. $T_f[j].\mathrm{ct} \longleftarrow T_f[j].\mathrm{ct} + 1$, $u_j \longleftarrow T_f[j].\mathrm{ct}$.
3. If $\perp = \mathcal{LK}(i\|\overline{u}_i)$, then $r_i \overset{\$}{\longleftarrow} \{0,1\}^{\kappa}$ and insert $(r_i, i, \overline{u}_i)$ into $\mathcal{LK}$, where $\overline{u}_i \longleftarrow T_w[i].\mathrm{ct}$ for $1 \leq i \leq m$.
4. $\mathbf{I}'[i,j] \longleftarrow RO(z_i)$, where $z_i \overset{\$}{\longleftarrow} \{0,1\}^{2\kappa}$ for $1 \leq i \leq m$.
5. Set $\mathbf{I}[i,j] \longleftarrow \mathbf{I}'[i,j]$ and $\mathbf{I}[i,j].\mathrm{st} \longleftarrow 1$ for $1 \leq i \leq m$.
6. If Query = $\langle\mathsf{Add}, |c|\rangle$, then simulate $c_j \longleftarrow \mathcal{E}.\mathsf{Enc}_k(\{0\}^{|c|})$ add c_j into $\mathcal{C}$, set $\tau_f \longleftarrow (\mathbf{I}', j)$ and output τ_f. Else, set $\tau_f' \longleftarrow (\mathbf{I}', j)$, remove c_j from $\mathcal{C}$ and output τ_f'.

Correctness and Indistinguishability of the Simulation: Given access pattern (τ_f, τ'_f) for a file f_{id}, $\mathcal{A}$ checks the correctness of update by searching all keywords $\mathcal{W} = \{w_{i_1}, \ldots, w_{i_l}\}$ in f_{id}. Since $\mathcal{S}$ is given access pattern $\Delta(\delta, \mathcal{F}, w, t)$ for a search query (which captures the last update before the search), the search operation always produces a correct result after an update (see *III-Correctness and Indistinguishability* argument). Hence, $\mathcal{S}$'s update tokens are correct and consistent.

It remains to show that (τ_f, τ'_f) have the correct probability distribution. In the real algorithm, the counter u_j is increased for each update as simulated in *IV-step 2*. If f_{id} is updated after the keyword w at row i is searched, a new r_i is generated for w as simulated in *IV-step 3* (r_i remains the same for consecutive updates but u_j increases). Hence, the real algorithm invokes $H(.)$ with a different $(r_i \| j \| u_j)$ for $1 \leq i \leq m$. $\mathcal{S}$ simulates this step by invoking $RO(\cdot)$ with z_i and $\mathbf{I}'[i, j] \longleftarrow RO(z_i)$, for $1 \leq i \leq m$. (τ_f, τ'_f) have random uniform distribution since $\mathbf{I}'$ has random uniform distribution and update operations are correct and consistent as shown above. c_j also has the correct distribution since Enc is an IND-CPA encryption. Hence, $\mathcal{A}$ does not abort during the simulation due to $\mathcal{S}$'s update tokens. The probability that $\mathcal{A}$ queries $RO(\cdot)$ on any z_i prior querying $\mathcal{S}$ on (τ_f, τ'_f) is negligible (i.e., $\frac{1}{2^{2\cdot\kappa}}$) and, therefore, $\mathcal{S}$ does not abort due to $\mathcal{A}$'s update query.

V. Final Indistinguishability Argument: (s_{w_i}, s_{id_j}, r_i) for $1 \leq i \leq m$ and $1 \leq j \leq n$ are indistinguishable from real tokens and keys since they are generated by PRFs that are indistinguishable from random functions. Enc is a IND-CPA scheme, the answers returned by $\mathcal{S}$ to $\mathcal{A}$for $RO(\cdot)$ queries are consistent and appropriately distributed, and all query replies of $\mathcal{S}$ to $\mathcal{A}$during the simulation are correct and indistinguishable as discussed in *I-IV Correctness and Indistinguishability* arguments. Hence, for all PPT adversaries, the outputs of $\mathbf{Real}_{\mathcal{A}}(\kappa)$ and $\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa)$ experiment are:

$$|\Pr[\mathbf{Real}_{\mathcal{A}}(\kappa) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}(\kappa) = 1]| \leq \mathsf{neg}(\kappa) \qquad \square$$

4.2 Privacy Levels

The leakage definition and formal security model imply various levels of privacy for different DSSE schemes. We summarize important privacy notions based on the various leakage characteristics discussed in [16, 27, 5, 24] with different levels of privacy as follows:

- *Size pattern*: The number of actual keyword-file pairs.
- *Forward privacy*: A search on a keyword w does not leak the identifiers of files matching this keyword for future files.
- *Backward privacy*: A search on a keyword w does not leak all historical update operations (e.g., addition /deletion) on the identifiers of files having this keyword.
- *Update privacy*: Update operation may leak different levels of information depending on the construction of the scheme. Specifically, we define five levels of update privacy, in which the level-1 leaks least information while the level-5 leaks the most, as follows:
  - Level-1 (*L1*) leaks only the time t of the update.
  - Level-2 (*L2*) leaks *L1* plus the identifier of the file being updated, the number of keywords in it and update type (i.e., add/delete/modify) (e.g., [27]).
  - Level-3 (*L3*) leaks *L2* plus if that identifier has same keywords added or deleted previously, and also when/if the same keywords were searched before (e.g., [5]).
  - Level-4 (*L4*) leaks *L3* plus if the same keyword was added/deleted from two files (e.g., [16]).

– Level-5 (*L5*) leaks the pattern of all intersections of everything is added/deleted and whether the keywords were searched for (e.g., [18]).

Corollary 1. *IM-DSSE framework offers forward-privacy.*

Proof (sketch). In IM-DSSE framework, the update involves reconstructing a new column/block of encrypted index **I**. The column/block is always encrypted with row keys that have never been revealed to the server (steps 2–4 in Simulate (τ_f, τ'_f))). This is achieved in IM-DSSE$^{\text{main}}$ and IM-DSSE$^{\text{I}}$ schemes by increasing the row counter after each keyword search operation (e.g., step 4 in Simulate (τ_w)) so that fresh row keys will always be used for subsequent update operations. In IM-DSSE$^{\text{II}}$ scheme, since all cryptographic operations are performed at the client side where no keys are revealed to the server, it is unable for the server to infer any information in the update, given that the encryption scheme is IND-CPA secure. These properties enable our IM-DSSE framework to achieve forward privacy. □

Corollary 2. *IM-DSSE^{II} and IM-DSSE^{I+II} achieve backward-privacy.*

Proof (sketch). In most DSSE schemes, the client sends a key that allows the server to decrypt a small part of the encrypted index during keyword search. The server can use this key to backtrack historical update operations on this part and therefore, compromise the backward-privacy. In IM-DSSE$^{\text{II}}$ and IM-DSSE$^{\text{I+II}}$ schemes, instead of sending the key to the server, the client requests this part and decrypts it locally. This prevents the server from learning information about historical update operations on the encrypted index and therefore, allows both schemes to achieve backward-privacy. □

One might observe that all developed DSSE schemes in IM-DSSE framework do not leak the update type (add/delete) on encrypted index **I** since it has the same access pattern on **I**. However, it can be distinguishable due to access pattern on encrypted files. This leakage can be sealed by sending a dummy file to the server during deletion. This strategy enables both add and delete operations to have the same procedure on both **I** and encrypted files and, therefore, achieves Level-1 of update privacy, where only the update time and a column index are leaked. Finally, since keyword-file relationships are represented by an encrypted incidence matrix, IM-DSSE framework also hides the *size pattern* (i.e., number of '1' entities in **I**), and therefore, achieve the size-obliviousness.

5 Performance Analysis and Evaluation

We evaluate the performance of our IM-DSSE framework in real-life networking and system settings. We provide a detailed cost breakdown analysis to fully assess criteria that constitute the performance overhead of our constructions. Given that such analysis is generally missing in the literature, this is the main focus of our performance evaluation. Finally, we give a brief asymptotic comparison of our framework with several DSSE schemes in the literature.

Implementation Details. We implemented our framework using C/C++. For cryptographic primitives, we used `libtomcrypt` cryptographic toolkit version 1.17. We modified low level routines to call AES hardware acceleration instructions (via Intel AES-NI library) if they are supported by the underlying hardware platform. Key generation was implemented using the expand-then-extract key generation paradigm analyzed in [19]. However, instead of using a standard hash function, we used AES-128 CMAC for performance reasons. This key derivation function has been formally analyzed and is standardized. Our random oracles were all implemented via 128-bit AES CMAC. For hash tables, we employed Google's C++ sparse hash map with the hash function being implemented by CMAC-based random oracles truncated to 80 bits. We implemented the IND-CPA encryption $\mathcal{E}$ using AES with CTR mode.

IM-DSSE framework contains the full implementation of all schemes presented in this article including IM-DSSE$^{\text{main}}$, IM-DSSE$^{\text{I}}$, IM-DSSE$^{\text{II}}$ and IM-DSSE$^{\text{I+II}}$, which can be freely accessible via our Github repository:

https://github.com/thanghoang/IM-DSSE/

Our implementation supports the encrypted index stored on either memory or local disk. Therefore, our schemes can be directly deployed in either storage-as-a-service (e.g., Amazon S3) or infrastructure-as-a-service clouds (e.g., Amazon EC2). For this experimental evaluation, we selected block cipher size $b = 128$ for IM-DSSE$^{\text{I}}$ and IM-DSSE$^{\text{I+II}}$ schemes.

Dataset. We used the Enron email dataset and select its subsets, ranging from 50,000 to 250,000 files with 240,000–940,000 unique keywords to evaluate the performance of our schemes with different encrypted index sizes. These selected sizes surpass the experiments in [18] by three orders of magnitude and are comparable to the experiments in [27].

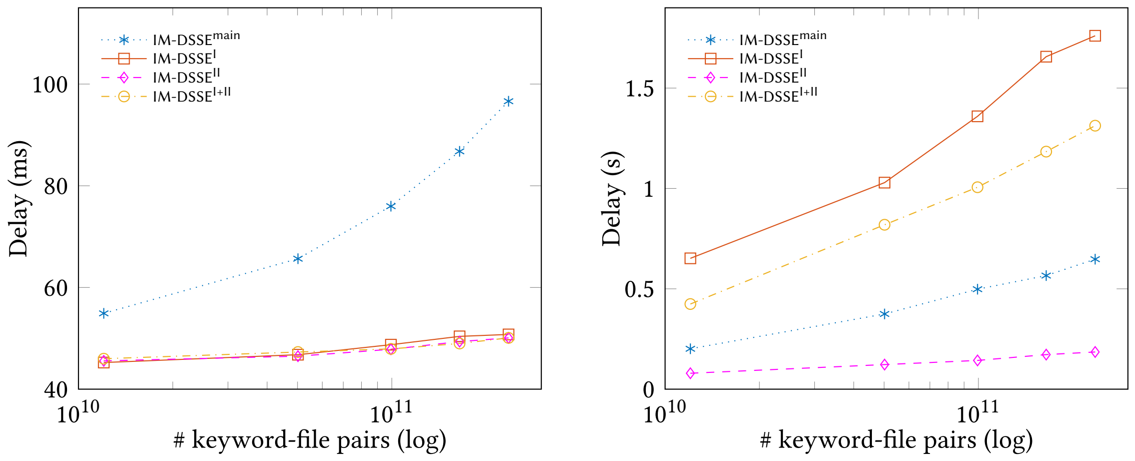
Hardware. We conducted the experiment with two settings:

(i) We used HP Z230 Desktop as the client and built the server using Amazon EC2 with `m4.4xlarge` instance type. The desktop was equipped with Intel Xeon CPU E3-1231v3 @ 3.40GHz, 16 GB RAM, 256 GB SSD and CentOS 7.2 installed. The server was installed with Ubuntu 14.04 and equipped with 16 vCPUs @2.4 GHz Intel Xeon E5-2676v3, 64 GB RAM and 500 GB SSD hard drive.

(ii) We selected LG G4 mobile phone to be the client machine, which runs Android OS, v5.1.1 (Lollipop) and is equipped with Qualcomm Snapdragon 808 64-bit Hexa-core CPU @1.8 GHz, 3GB RAM and 32 GB internal storage. Notice that AES-NI library cannot be used to accelerate cryptographic operations on this mobile device since its incompatible CPU, which affects the performance of our schemes in the mobile environment as will be shown in the following section.

We disabled the slow-start TCP algorithm and maximized initial congestion window parameters in Linux (i.e., 65535 bytes) (see [10] for more insights) to reduce the network impact during the initial phase in case the scheme requires low amount of data to be transmitted.

Performance Results. Figure 1 presents the overall performance in terms of end-to-end cryptographic delay of all schemes in IM-DSSE framework. In this experiment, we located client and server in the same geographical region, resulting in the network latency of 11.2 ms and throughput of 264 Mbps. We refer to this configuration as a *fast network* setting. Notice that we only measured the delay due to accessing the encrypted index $\mathbf{I}$, and omitted the time to access encrypted files (i.e., set $\mathcal{C}$) as it is identical for all (D)SSE and non-SSE schemes. For instance, in keyword search, we measured the delay of IM-DSSE$^{\text{main}}$ scheme and IM-DSSE$^{\text{I}}$ scheme by the time the client sends the request and the server finishes decrypting an entire row of the encrypted index and gets cells whose value is 1. The IM-DSSE$^{\text{main}}$ scheme and its extended versions took less than 100 ms to perform a keyword search, while it took less than 2 seconds to update a file. The cost per keyword search depends linearly on the maximum number of files in the database (i.e., $\mathcal{O}(n)$) and yet it is highly practical even for very large numbers of keyword-file pairs (i.e., more than 10^{11} pairs). Indeed, we confirm that the search operation in IM-DSSE is very fast and most of the overhead is due to network communication delay as it will be later analyzed in this section. Note that the costs for adding and deleting files (updates) over the encrypted index are similar since their procedure is identical. The keyword search operation delay of IM-DSSE$^{\text{main}}$ is higher than that



(a) Keyword search

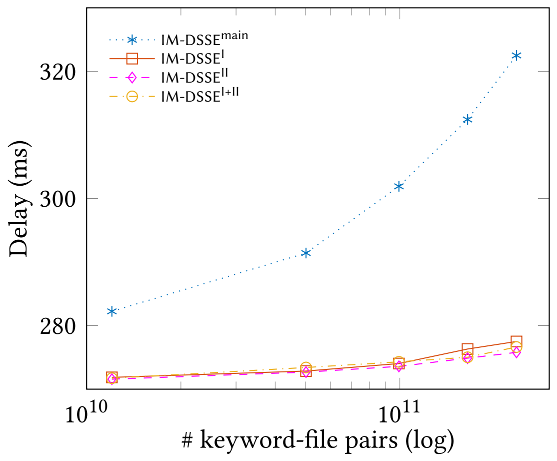
(b) File update

Figure 1: The latency of our schemes with fast network.

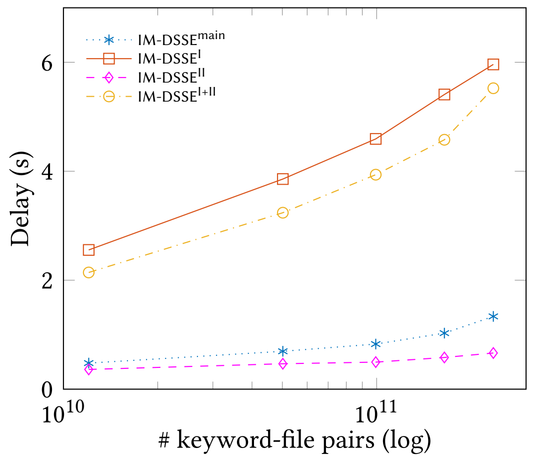
of extended schemes and it increases as the size of the encrypted index increases due to two reasons: First, the encrypted index **I** in IM-DSSE$^{\text{main}}$ scheme is bit-by-bit encrypted compared with 128-bit block encryption in IM-DSSE$^{\text{I}}$. Hence, the server needs to derive more AES keys than in IM-DSSE$^{\text{I}}$ to decrypt a whole row. Thus, the gap between IM-DSSE$^{\text{main}}$ and IM-DSSE$^{\text{I}}$ represents the server computation cost required for this key derivation and encryption. Second, processes in IM-DSSE$^{\text{main}}$ scheme are performed subsequently, in which the server needs to receive some information sent from the client first before being able to derive keys to decrypt a row. Such processes in IM-DSSE$^{\text{I}}$ and IM-DSSE$^{\text{II}}$ can be parallelized, where the client generates the AES-CTR keys while receiving a row of data transmitted from the server. We can see that the delay is not so different between IM-DSSE$^{\text{I}}$ and IM-DSSE$^{\text{II}}$ and IM-DSSE$^{\text{I+II}}$. This indicates that using 128-bit encryption significantly reduces the server computation cost to be negligible as it will be later shown.

Considering the file update operation, our IM-DSSE$^{\text{main}}$ and IM-DSSE$^{\text{II}}$ schemes leverage 1-bit encryption and, therefore, it does not require to transfer a 128-bit block to the client first prior to updating the column as in IM-DSSE$^{\text{I}}$ and IM-DSSE$^{\text{I+II}}$ schemes. Hence, they are faster and less affected by the network latency than IM-DSSE$^{\text{I}}$ and IM-DSSE$^{\text{I+II}}$. So, the gap between such schemes reflects the data download delays, which will be significantly higher on slower networks as shown in the next experiment. Update in IM-DSSE$^{\text{I+II}}$ is considerably faster than in IM-DSSE$^{\text{I}}$ because it allows for parallelization, in which the client can pre-compute AES-CTR keys while receiving data from the server. In IM-DSSE$^{\text{I}}$, such keys cannot be computed as they need some information being sent from the server beforehand (i.e., state data $\mathbf{I}[*, j].\mathsf{st}$).

The impact of network quality. The previous experiments were conducted on a high-speed network, which might not be widely available in practice. Hence, we additionally investigated how our schemes perform when the network quality is degraded. To do that, we setup the server to be geographically located distant from the client machine, resulting in the network latency and throughput to be 67.5 ms and 46 Mbps, respectively . Figure 2 shows the end-to-end crypto delay of our schemes in this *moderate network* setting. Due to the high network latency, search operation of each scheme is slower than that of fast network by 230ms. The impact of the network latency is clearly shown in the update operation as reflected in Figure 2b. The delays of IM-DSSE$^{\text{I}}$, IM-DSSE$^{\text{I+II}}$ are significantly higher than those of IM-DSSE$^{\text{main}}$ and IM-DSSE$^{\text{II}}$. As explained previously, this gap actually reflects the download delay incurred by such schemes.

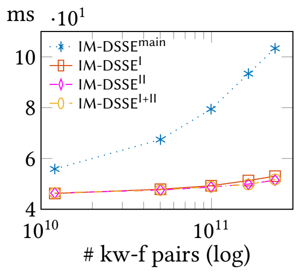


(a) Keyword search

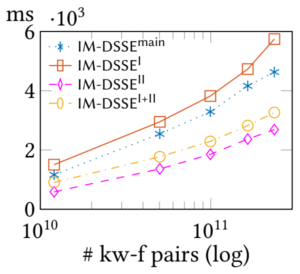


(b) File update

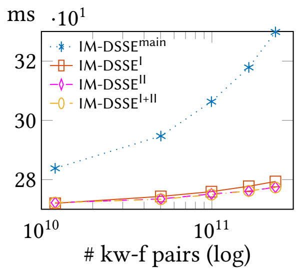
Figure 2: The latency of our schemes with moderate network.



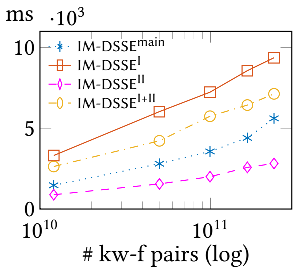
(a) Keyword search



(b) File update



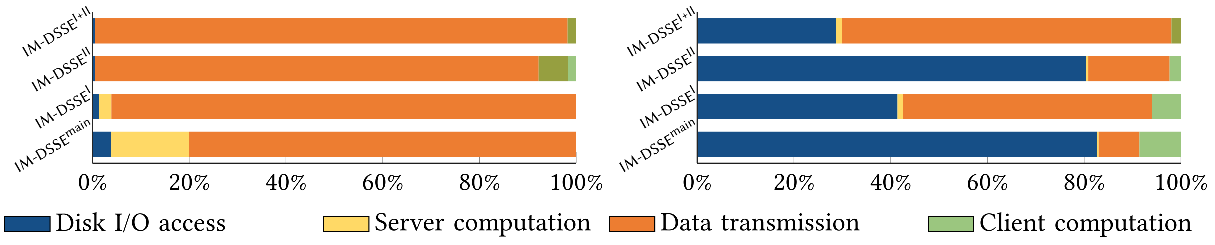
(c) Keyword search



(d) File update

Figure 3: The latency of IM-DSSE framework with SSD server-storage and (a,b) fast and (c,d) moderate networks.

Storage location of encrypted index: RAM vs. disk. Another important performance factor for DSSE is the encrypted index storage access delay. Hence, we investigated the impact of the encrypted index storage location on the performance of our schemes. Clearly, the ideal case is to store all server-side data on RAM to minimize the delay introduced by storage media access as shown in previous experiments. However, deploying a cloud server with a very large amount of RAM capacity can be very costly. Thus, in addition to the RAM-stored results shown previously, we stored the encrypted index on the secondary storage unit (i.e., SSD drive), and then measured how overall delays of our scheme were impacted by this setting. Figure 3 presents results with two aforementioned network quality environments (i.e., fast and moderate speeds). In IM-DSSE$^{\text{main}}$ and IM-DSSE$^{\text{I}}$ schemes, the disk I/O access is incurred by loading a part of the encrypted index including value $\mathbf{I}.v$ and state $\mathbf{I}.\text{st}$. It is clear that the disk I/O access time incurred an insignificant latency to the overall delay in terms of keyword search operation as shown in Figures 3a and 3d since our schemes achieve perfect locality as defined by Cash et al. [7]. However, in the file update operation, the delay in IM-DSSE framework was 1–4 seconds more, compared with RAM-based storage. That is because we stored all cells in each row of the encrypted matrix $\mathbf{I}$ in contiguous memory blocks. Therefore, keyword search invokes accessing *subsequent memory blocks* while update operation results in accessing *scattered blocks* which incurs much higher disk I/O access time. Due to the incidence matrix data structure and this storage strategy, our search operation was not affected as much by disk I/O access time as other non-local DSSE schemes (e.g., [5, 6, 17]), which require accessing random memory blocks for security.



(a) Keyword search (b) File update

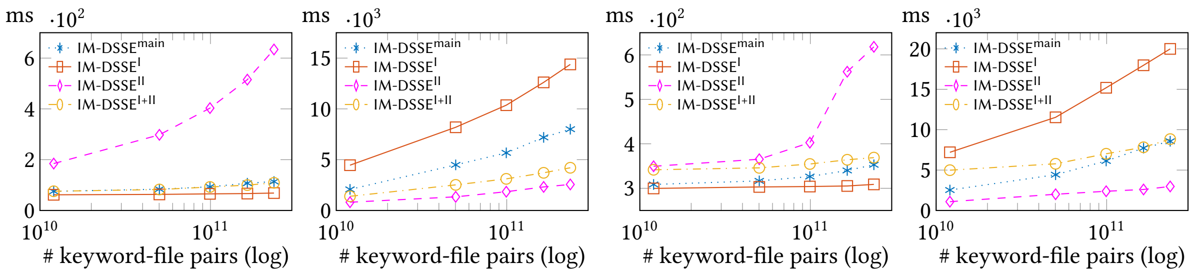
Figure 4: Detailed costs of IM-DSSE framework with moderate network and SSD server-storage.

Cost breakdown. We dissected the overall cost of our schemes previously presented in Figures 1, 2 and 3 to investigate which factors contribute a significant amount to the total delay of each scheme. For analysis, we selected the cost of our schemes when performing on the largest encrypted index size being experimented (i.e., 2.36×10^{11}) with moderate network speed, where the encrypted index is stored on a SSD drive. Figure 4 presents the major factors that contribute to the total delay of our schemes during keyword search and file update operations.

Considering the search operation, it is clearly data transmission what occupied the largest amount of delay of all schemes. In our IM-DSSE$^{\text{main}}$ and IM-DSSE$^{\text{I}}$ schemes, most of the computations were performed by the server wherein cryptographic operations were accelerated by AES-NI so that they only took a small number of the total, especially in IM-DSSE$^{\text{I}}$ scheme. Meanwhile, the client only performed simple computations such as search token generation so that its cost was negligible. In IM-DSSE$^{\text{II}}$ and IM-DSSE$^{\text{I+II}}$ schemes, encrypted data were decrypted at the client side, while the server did nothing but transmission. Therefore, the client computation cost took a small portion of the total delay and the server's cost was negligible. However, as indicated in Section 3, the client computation and data transmission in IM-DSSE$^{\text{II}}$ and IM-DSSE$^{\text{I+II}}$ are fully parallelizable where their partially parallel costs are indicated by their overlapped area in Figure 4a. Hence, we can infer that client computation was actually dominated by data transmission and, therefore, the computation cost did not affect the total delay of the schemes. As explained above, we stored the encrypted matrix on disk with row-friendly strategy so that the disk I/O access time due to keyword search was insignificant, which contributed less than 3% to the total delay.

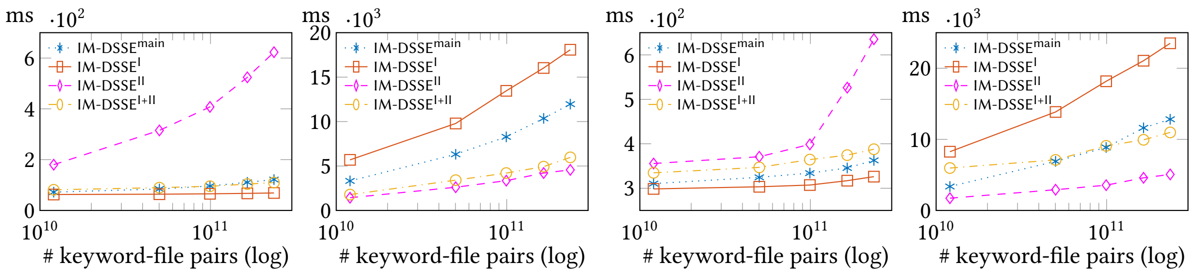
In contrast, it is clear that disk I/O access time occupied a considerable proportion of the overall delay of the update operation, especially in the IM-DSSE$^{\text{main}}$ and IM-DSSE$^{\text{II}}$ schemes due to non-contiguous memory access. Data transmission was the second major factor contributing to the total delay. As the server did not perform any expensive computations, its cost was negligible in all schemes. The client performed cryptographic operations which were accelerated by AES-NI library so that it only contributed less than 7% of the overall cost. Additionally, the client computation was mostly parallelized with the data transmission and the server's operations in IM-DSSE$^{\text{II}}$ and IM-DSSE$^{\text{I+II}}$ schemes so that it can be considered not to significantly impact the total delay.

Realization on mobile environments. Finally, we evaluated our schemes' performance when deployed on a mobile device with limited computational resources. Similarly to the desktop experiments, we tested on fast and moderate network speed by geographically locating the server close and far away from the mobile phone, respectively. The phone was connected to a local WiFi which, in turn, allowed the establishment of the connection to the server via a wireless network resulting in the network latency and throughput of fast network case to be 18.8 ms, 136 Mbps while those of moderate case were 76.3 ms and 44 Mbps respectively. Figures 5 and 6 present the benchmarked results with aforementioned net-



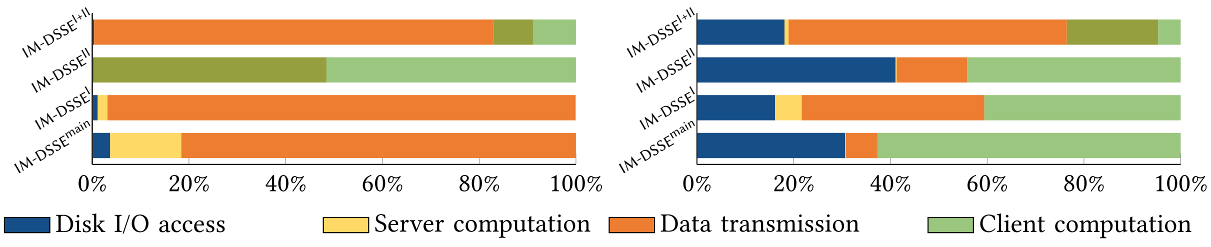
(a) Keyword search (b) File update (c) Keyword search (d) File update

Figure 5: The latency of IM-DSSE framework on mobile and RAM server-storage with (a,b) fast and (c,d) moderate networks.



(a) Keyword search (b) File update (c) Keyword search (d) File update

Figure 6: The latency of IM-DSSE framework on mobile and SSD server-storage with (a,b) fast and (c,d) moderate networks.



(a) Keyword search (b) File update

Figure 7: Detailed costs of IM-DSSE framework on mobile with moderate network and SSD server-storage.

work settings when the data in the server were stored on primary (i.e., RAM) and secondary (i.e., SSD) storage units, respectively. In the mobile environment, the IM-DSSE^{II} scheme performed considerably slower than others in terms of keyword search. That is because, in this scheme, a number of cryptographic operations (i.e., $\mathcal{O}(n)$) were performed by the mobile device. Moreover, these operations were not accelerated by AES-NI library as in our Desktop machine because the mobile CPU did not have special crypto accelerated instructions. Considering the keyword search performance of IM-DSSE^{II} in the moderate network setting (i.e., Figures 5c and 6c), we can see that its delay significantly increased when the size of encrypted index exceeded 10^{11} keyword-file pairs. This is because starting from this size of the encrypted index, the client computation began to dominate the data transmission cost. The update delays of our schemes, especially the IM-DSSE^{main} and IM-DSSE^{II} schemes, were substantial in the mobile environment because the mobile platform had to perform intensive cryptographic operations.

Figure 7 shows the decomposition of the total end-to-end delay of our schemes in the out-of-state

Table 1: Security and asymptotic complexity of some state-of-the-art DSSE schemes.

Scheme/Property	KPR12 [18]	KP13 [16]	CJK⁺14 [5]	HK14 [12]	SPS14[27]	B16 [3]	**IM-DSSE**
Update Privacy	*L5*	*L4*	*L3*	*L3*	*L2*	*L2*	***L1***
Size Privacy	✗	✗	✗	✗	✗	✗	✓
Forward Privacy	✗	✗	✗	✗	✓	✓	✓
Backward Privacy	✗	✗	✗	✗	✗	✗	✓†
Client Storage	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(m')$	$\mathcal{O}(N'^{\alpha})$	$\mathcal{O}(m \log n)$	$\mathcal{O}(m+n)$
Index Size	$\mathcal{O}(N'+m)$	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(N')$	$\mathcal{O}(N')$	$\mathcal{O}(N')$	$\mathcal{O}(N')$	$\mathcal{O}(m \cdot n)$
Search Cost	$\mathcal{O}(r)$	$\mathcal{O}\left(\frac{r \log n}{p}\right)$	$\mathcal{O}\left(\frac{r+d_w}{p}\right)$	$\mathcal{O}(r)$	$\mathcal{O}\left(\min\left\{\begin{matrix} d_w + \log N' \\ r \log^3 N' \end{matrix}\right\}\right)$	$\mathcal{O}(r+d_w)$	$\mathcal{O}\left(\frac{n}{p}\right)$
Update Cost	$\mathcal{O}(m'')$	$\mathcal{O}(m \log n)$	$\mathcal{O}(m''+r'')$	$\mathcal{O}(m'')$	$\mathcal{O}\left(m'' \log^2 N'\right)$	$\mathcal{O}(m'')$	$\mathcal{O}(m)$
Parallelizable	✗	✓	✓	✗	✗	✗	✓

† IM-DSSE$^{\mathrm{II}}$ and IM-DSSE$^{\mathrm{I+II}}$ schemes in our IM-DSSE framework offer backward-privacy (see Section 4).
• m and n denote the maximum number of keywords and files, respectively. $m' < m$ and $n' < n$ denote the actual number of keywords and files, respectively. $N' \leq m' \cdot n'$ is # keyword-file pairs. m'' = # unique keywords included in an updated file, r = # files matching search query, p = # processors, $0 < \alpha < 1$, d_w = # historical update (add/delete) operations on keyword w, r'' = (accumulated) # unique keywords being newly added.
• We omitted the security parameter κ for analyzed complexity cost. Update privacy levels $\{L1, \ldots, L5\}$ are described in Section 4.

network setting when the server data were stored on a SSD hard drive. For the search operation, the detailed costs of IM-DSSE$^{\mathrm{main}}$ and IM-DSSE$^{\mathrm{I}}$ schemes are the same as those of the desktop setting because computations were performed by the server while the client only did lightweight computation to generate the search token. In the IM-DSSE$^{\mathrm{II}}$ scheme, the client computation contributed almost 100% to the total delay due to $\mathcal{O}(n)$ number of AES-CTR decryptions, compared with that of $\mathcal{O}(n)/128$ in IM-DSSE$^{\mathrm{I+II}}$ which was all dominated by the data transmission delay. The limitation of computational capability of the mobile device is reflected clearly in Figure 7b, wherein the client computation cost accounted for a considerable amount of the overall delay of most schemes except for IM-DSSE$^{\mathrm{I+II}}$ scheme.

Asymptotic performance comparison. Compared to Kamara et al. in [18], which achieves optimal sublinear search time but leaks significant information for update, our IM-DSSE framework has linear search time but achieves highly secure updates. Moreover, the scheme in [18] can not be parallelized whereas our schemes can be. Kamara et al. in [16] relies on red-black trees as the main data structure, achieves parallel search and oblivious update. However, it incurs extreme server storage overhead due to its large encrypted index size. The scheme of Stefanov et al. [27] requires relatively high client storage (e.g., 210 MB for moderate size of file-keyword pairs), where the client fetches a non-negligible amount of data from the server and performs an oblivious sort on it. We only require two hash tables and three symmetric secret keys storage. The scheme in [27] also requires significant amount of data storage (e.g., 2000–3200 bits) per keyword-file pair at the server side versus 1-2 bits per keyword-file pair along with a hash table in our framework. The data structure in [27] grows linearly with the number of deletion operations, which requires re-encrypting the data structure eventually. Our schemes do not require re-encryption (but we assume an upper bound on the maximum number of files), and our storage is constant regardless of the number of updates. Cash et al. introduced the most efficient DSSE scheme [5] which achieves a sub-linear search complexity. Despite being asymptotically better than our scheme, our simulated result showed that, it is only one order of magnitude faster in practice on very-large databases with hundred millions of documents (as used in [5]). Remark that this scheme does not offer forward-privacy and therefore, it is less secure than our schemes. Table 1 provides a asymptotic comparison of our framework with several prominent DSSE schemes regarding security, asymptotic complexity and some additional properties.

6 Related Work

SSE was first introduced by Song et al. [26] and it was followed by several improvements (e.g., [9, 8]). Curtmola et al. [9] proposed a sublinear SSE scheme and introduced the security notion for SSE called *adaptive security against chosen-keyword attacks (CKA2)*. Refinements of [9] have been proposed which offer extended functionalities (e.g., [29]). However, the static nature of those schemes limited their applicability to applications required dynamic file collections. Kamara et al. were among the first to develop a DSSE scheme in [18] that could handle dynamic file collections via an encrypted index. However, it leaks significant information for updates and it is not parallelizable. Kamara et al. [16] proposed a DSSE scheme, which leaked less information than that of [18] and it was parallelizable. Recently, a series of new DSSE schemes (e.g., [27, 5, 24, 12, 3, 15]) have been proposed which offer various trade-offs between security, functionality and efficiency properties such as small leakage (e.g., [27]), scalable searches with extended query types (e.g., [6, 30, 15]), or high efficiency (e.g., [24, 21]). Among these state-of-the art DSSE schemes, we have already selected several typical ones (i.e.,[27, 5, 18, 17, 3, 12]) and compared their security and efficiency properties in Table 1 and Section 5. Inspired by the work from [5], Kamara et al. in [15] proposed a new DSSE scheme which supports more complex queries such as disjunctive and boolean queries and achieved sub-linear search time.

Due to the deterministic keyword-file relationship, most traditional DSSE schemes (including ours) leak search and access patterns defined in Section 4 which are vulnerable to statistical inference attacks. A number of attacks (e.g., [14, 25, 4, 20, 23, 32]) have been demonstrated. Several DSSE schemes have been proposed to deal with such leakages (e.g., [13, 2]) but they are neither efficient nor fully secure. Oblivious Random Access Machine (ORAM) (e.g., [11, 28]) can hide search and access patterns in DSSE. Despite a lot of progress on these techniques, their costs are still extremely high to be applied to DSSE in practice [22]. Hence, hiding access pattern leakages in DSSE with more efficient approaches is still an open research problem that needs to be resolved to make DSSE fully secure in real-world applications.

7 Conclusions

In this article, we presented IM-DSSE, a new DSSE framework which offers very high privacy, efficient updates, low search latency simultaneously. Our constructions rely on a simple yet efficient incidence matrix data structure in combination with two hash tables that allow efficient and secure search and update operations. Our framework offers various DSSE construction, which are specifically designed to meet the needs of cloud infrastructure and personal usage in different applications and environments. All of our schemes in IM-DSSE framework are proven to be secure and achieve the highest privacy among their counterparts. We conducted a detailed experimental analysis to evaluate the performance of our schemes on real Amazon EC2 cloud systems. The achieved results showed the high practicality of our framework even when deployed on mobile devices with large datasets. We released the full-fledged implementation of our framework for public use and analysis.

References

[1] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and Communications Security (CCS '93)*, pages 62–73, NY, USA, 1993. ACM.

[2] C. Bösch, A. Peter, B. Leenders, H. W. Lim, Q. Tang, H. Wang, P. Hartel, and W. Jonker. Distributed searchable symmetric encryption. In *Privacy, Security and Trust (PST), 2014 Twelfth Annual International Conference on*, pages 330–337. IEEE, 2014.

[3] R. Bost. Sophos âĂŞ forward secure searchable encryption. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*. ACM, 2016.

[4] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM CCS*, pages 668–679, 2015.

[5] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawcyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *21th Annual Network and Distributed System Security Symposium — NDSS 2014*. The Internet Society, February 23-26, 2014.

[6] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology, CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 353–373, 2013.

[7] D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, pages 351–368. Springer, 2014.

[8] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594, 2010.

[9] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 79–88. ACM, 2006.

[10] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing tcp's initial congestion window. *Computer Communication Review*, 40(3):26–33, 2010.

[11] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[12] F. Hahn and F. Kerschbaum. Searchable encryption with secure and efficient updates. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 310–320. ACM, 2014.

[13] T. Hoang, A. Yavuz, and J. Guajardo. Practical and secure dynamic searchable encryption via oblivious access on distributed data structure. In *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*. ACM, 2016.

[14] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, volume 20, page 12, 2012.

[15] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. *EUROCRYPT 2017*, 2017.

[16] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security (FC)*, volume 7859 of *Lecture Notes in Computer Science*, pages 258–274. Springer Berlin Heidelberg, 2013.

[17] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security*, pages 258–274. Springer, 2013.

[18] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 965–976, New York, NY, USA, 2012. ACM.

[19] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Advances in Cryptology - CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, August 15-19 2010.

[20] C. Liu, L. Zhu, M. Wang, and Y.-a. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences*, 265:176–188, 2014.

[21] I. Miers and P. Mohassel. Io-dsse: Scaling dynamic searchable encryption to millions of indexes by improving locality. 2016.

[22] M. Naveed. The fallacy of composition of oblivious ram and searchable encryption. *IACR Cryptology ePrint Archive*, 2015:668, 2015.

[23] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM CCS*, pages 644–655, 2015.

[24] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. In *35th IEEE Symposium on Security and Privacy*, pages 48–62, May 2014.

[25] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*. ACM, 2016.

[26] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[27] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *21st Annual Network and Distributed System Security Symposium — NDSS 2014*. The Internet Society, February 23-26, 2014.

[28] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications security*, pages 299–310. ACM, 2013.

[29] B. Wang, S. Yu, W. Lou, and Y. T. Hou. Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud. In *INFOCOM, 2014 Proceedings IEEE*, pages 2112–2120. IEEE, 2014.

[30] Z. Xia, X. Wang, X. Sun, and Q. Wang. A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):340–352, 2016.

[31] A. A. Yavuz and J. Guajardo. Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. In *International Conference on Selected Areas in Cryptography*, pages 241–259. Springer, 2015.

[32] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *25th USENIX Security '16*, pages 707–720, Austin, TX, 2016.