Toward Reliable and Secure Cloud Services with Fault-Tolerant Searchable Encryption

Cong Zuo, *Member*, IEEE, Bingjing Wang, Jianghua Liu, Shujie Cui, *Member*, IEEE, Jun Shao, *Senior Member*, IEEE, Huaxiong Wang, *Senior Member*, IEEE, Liehuang Zhu, *Senior Member*, IEEE, Giovanni Russello

***Abstract*—Dynamic Searchable Symmetric Encryption (DSSE) plays a crucial role in secure cloud-based database systems, as it enables efficient keyword search and dynamic updates over encrypted data. However, practical deployment of DSSE schemes faces two significant challenges. First, clients may inadvertently perform faulty updates—such as re-adding an existing keyword-identifier pair or attempting to delete a non-existent one—which can compromise the correctness of subsequent search results. Second, even with correctly issued updates, malicious servers may return incorrect or incomplete search results, undermining data integrity. To address these challenges, we propose `FVDSSE`, the first fault-tolerant DSSE scheme that tolerates client-side operational faults and provides result verifiability against malicious servers. Moreover, it simultaneously ensures strong privacy by guaranteeing forward and backward privacy—two essential properties for any practical DSSE. To further optimize performance, we present `FVDSSE-C`, an enhanced variant that leverages caching techniques. Experimental evaluations on a real-world dataset show that `FVDSSE-C` achieves up to $130\times$ improvement in search efficiency and $3\times$ reduction in communication overhead compared to the state-of-the-art scheme (`YCR22-C`).**

***Index Terms*—Fault-tolerance, Verifiability, Forward Privacy, Backward Privacy, Dynamic Searchable Symmetric Encryption.**

I. Introduction

With the widespread adoption of cloud storage services (e.g., Google Drive [1], OneDrive [2]), a critical requirement for privacy-preserving data outsourcing has emerged: enabling secure and efficient query processing on encrypted data [3]–[5]. As a fundamental enabling technology, Dynamic Searchable Symmetric Encryption (DSSE) [6] allows clients to perform keyword searches and dynamic updates on encrypted databases while preserving data confidentiality. These requirements are particularly critical for cloud database services—a core category of cloud services where data is persistently stored, indexed, and queried [7]–[9]. However, the practical deployment of DSSE in such services faces two fundamental trust challenges, as outlined in the abstract: (1) ensuring result verifiability against potentially malicious service providers, and (2) achieving fault tolerance against common client-side operational errors.

Existing research has made significant progress in addressing specific aspects of these challenges. For instance, to counter malicious servers, Verifiable DSSE (VDSSE) schemes [10]–[12] enable clients to cryptographically verify the integrity and completeness of search results. To mitigate information leakage from update patterns, forward and backward private DSSE schemes [13], [14] limit information leakage from query and update patterns, offering protection against inference attacks. However, a crucial and often dominant risk in operational settings stems from client-side operational faults, such as duplicate additions or invalid deletions. Industry reports indicate that client-side operational faults constitute over 80% of data incidents in managed cloud services [15], highlighting the need for mechanisms that can tolerate such faults without compromising system correctness.

As discussed above, current research ignores the fact that client-side operational faults will have a negative impact on the practical deployment of DSSE in real-world cloud service. This neglect is the root cause of the limitations in existing schemes. As summarized in Table I, schemes that provide strong verifiability often lack fault tolerance, causing correct results to be rejected when clients make operational faults. Conversely, fault-tolerant schemes frequently sacrifice either verifiability or privacy guarantees. For instance, `YCR22-C` supports verifiability but lacks backward privacy, while `ROSE` provides fault tolerance but forfeits verifiability. This separation of capabilities creates a critical gap for cloud database services that require simultaneous assurance of three essential properties: (1) fault tolerance against client operational faults, (2) result verifiability to ensure trustworthy query results, and (3) strong privacy through forward and backward privacy.

To bridge this gap, we introduce FVDSSE—the first Fault-tolerant Verifiable DSSE scheme that simultaneously achieves fault tolerance, result verifiability, and strong privacy while maintaining practical efficiency for large-scale deployments. Our approach shows that these properties can be integrated

Manuscript received XX Month 202X; revised XX Month 202X; accepted XX Month 202X. Date of publication XX Month 202X; date of current version XX Month 202X. This research was supported in part by the Beijing Natural Science Foundation (Grant No. L251002), the National Natural Science Foundation of China (Grant No. 62372040, 62572045, 62202226, 62272413), and the Project funded by China Postdoctoral Science Foundation (2023T160317).
Cong Zuo and Liehuang Zhu are with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China, and also with the Shandong Key Laboratory of Energy Industry Internet Big Data Technology, Jinan 250003, Shandong, China (e-mail: zuocong10@gmail.com, liehuangz@bit.edu.cn). Bingjing Wang is with the School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing 100081, China (e-mail: bjwang@bit.edu.cn). Jianghua Liu is with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China (e-mail: jhliu@njust.edu.cn). Shujie Cui is with the Faculty of Information Technology, Monash University, Melbourne, VIC 3800, Australia (e-mail: shujie.cui@monash.edu). Jun Shao is with the School of Computer Science and Technology, Zhejiang Gongshang University, Hangzhou 310018, China (e-mail: chn.junshao@gmail.com). Huaxiong Wang is with the School of Physical & Mathematical Sciences, Nanyang Technological University, Singapore 639798 (e-mail: hxwang@ntu.edu.sg). Giovanni Russello is with the School of Computer Science, University of Auckland, Auckland 1010, New Zealand (e-mail: g.russello@auckland.ac.nz). *(Corresponding author: Jianghua Liu.)*

TABLE I: Comparison with related works

Scheme	Fault-tolerant	Verifiability	FP	BP	Client Storage	Computation Search	Computation Update	Communication Search	Communication Update	Roundtrips
Moneta [14]	✓	✗	✓	Type-I	$O(1)$	$\tilde{O}(u_w log N + log^3 N)$	$\tilde{O}(log^2 N)$	$\tilde{O}(u_w log N + log^3 N)$	$\tilde{O}(log^3 N)$	3
Fides [14]	✓	✗	✓	Type-II	$O(\|\mathbf{W}\|log\|D\|)$	$O(u_w)$	$O(1)$	$O(u_w)$	O(1)	2
FB-DSSE [18]	✗	✗	✓	Type-I$^-$	$O(\|\mathbf{W}\|log\|D\|)$	$O(u_w)$	$O(1)$	$O(1)$	$O(1)$	1
ROSE [19]	✓	✗	✓	Type-III	$O(\|\mathbf{W}\|log\|D\|)$	$O((n_w + s_w + 1)d_w)$	$O(1)$	$O(n_w)$	$O(1)$	2
YCR22-C [20]	✓	✓	✓	✗	$O(\|\mathbf{W}\|log\|D\|)$	$O(u_w + 1)$	$O(1)$	$O(u_w + n_w)$	$O(1)$	1
FVDSSE	✓	✓	✓	Type-I$^-$	$O(\|\mathbf{W}\|log\|D\|)$	$O(u_w)$	$O(1)$	$O(u_w)$	$O(1)$	1
FVDSSE-C	✓	✓	✓	Type-I$^-$	$O(\|\mathbf{W}\|log\|D\|)$	$O(u'_w + 1)$	$O(1)$	$O(u'_w)$	$O(1)$	2

N is the number of keyword-identifier pairs. u_w is the number of updates for keyword w, and u'_w is the number of updates after the previous search query for keyword w. d_w is the number of deleted entries for keyword w, and n_w is the number of files that currently matching keyword w ($n_w = u_w - d_w$). d_{max} is the maximum number of files that the scheme can delete. s_w is related to the number of occurred search queries. $|\mathbf{W}|$ is the collection of distinct keywords, and $|D|$ is the total number of files. $\tilde{O}$ notation hides polylogarithmic factors. **FP** and **BP** stand for forward privacy and backward privacy, respectively.

effectively rather than treated as conflicting goals. As shown in Table I, FVDSSE provides fault tolerance, result verifiability, forward privacy, and backward privacy, addressing the limitations of prior work. By employing only symmetric cryptographic primitives and novel state management techniques, FVDSSE provides this complete protection without relying on public-key operations or specialized hardware, making it suitable for practical, cost-sensitive cloud deployments.

Our Contributions:

- We propose FVDSSE, the first DSSE scheme that simultaneously provides fault tolerance, verifiability, forward privacy, and Type-I$^-$ backward privacy. By employing only symmetric cryptographic primitives, FVDSSE achieves these properties without the overhead of public-key operations or authenticated encryption, presenting a practical solution to the DSSE integration gap identified in prior work.
- To address performance requirements in practical cloud services, we design FVDSSE-C, an enhanced variant that incorporates intelligent caching strategies. FVDSSE-C optimizes data locality [16], [17] and eliminates redundant processing for repeated queries through carefully designed cache management, significantly improving search efficiency while maintaining all security guarantees for enterprise workloads with recurring query patterns.
- We implement both schemes within a realistic cloud service framework and conduct comprehensive evaluations on real-world datasets. Compared to the state-of-the-art fault-tolerant verifiable scheme YCR22-C, our FVDSSE-C achieves up to 130× improvement in search efficiency and reduces communication overhead by 3×, while providing stronger privacy guarantees. These performance improvements translate directly to reduced operational costs and enhanced service quality—critical factors for cloud service adoption decisions.

II. Related Work

With the escalating volume of data outsourced to the cloud, trustworthy query services have become essential for practical applications ranging from business analytics to healthcare systems [21], [22]. Recent research has explored various aspects of cloud service security, including verifiable fuzzy search, efficiency-security trade-offs, and fraud detection in emerging network paradigms [23]–[25]. As a fundamental building block in this domain, Searchable Symmetric Encryption (SSE) provides the cryptographic primitive for encrypted data retrieval. Early works [26], [27] established the foundational security definitions and efficiency improvements, with the deployment of inverted indexes achieving sub-linear search time—a critical advancement for practical cloud-based big data applications. Subsequent research extended SSE to support rich queries [28]–[31], multi-client settings [32], [33], and improved locality [16], [17], progressively aligning these schemes with modern cloud service requirements. However, these early constructions primarily target static datasets.

The transition to dynamic data operations, essential for modern cloud applications, led to Dynamic SSE (DSSE) [6].But update operations introduced new leakage patterns that could be exploited by attackers in cloud settings [34]. To alleviate these attacks, DSSE schemes are required to maintain forward and backward privacy, which were first informally introduced by Stefanov et al. [35]. In particular, forward privacy ensures that the server cannot match the new updates with the previously search queries, and backward privacy guarantees that the files previously added and later deleted cannot be learned by the server. Later, Bost [13] formally gave the forward privacy definition and a concrete forward private DSSE. The formal backward privacy is first introduced by Bost et al. [14]. Later, many DSSE schemes supporting forward privacy and backward privacy have been introduced [14], [18], [36]–[39], thereby enhancing privacy protection for cloud-based database services.

The integrity of search results is another critical requirement for trustworthy cloud database services, particularly in environments where the service provider is not fully trusted. Verifiable SSE (VSSE) schemes address this need by incorporating cryptographic proof structures that enable clients to verify the correctness and completeness of results returned by the server. Early works by Kurosawa et al. [40] introduced a UC-secure SSE with verifiability, while Chai et al. [41] achieved verifiability through an encrypted PPTrie construction[1], though their scheme was limited to single-keyword queries. To support more complex query patterns prevalent in cloud services, Wang et al. [42] developed a verifiable SSE based on the OXT [28] framework, which supports conjunctive queries. A

[1]PPTrie [41] is a tree-based indexing structure enabling verifiability via prefix signature hashing and encrypted bitmap techniques.

key limitation of these early schemes is their static nature; they cannot accommodate dynamic updates, thus restricting their practicality for evolving cloud datasets.

Extending verifiability to dynamic settings, resulting in Verifiable DSSE (VDSSE), introduces the challenge of handling incorrect results stemming from two primary sources: a malicious server returning faulty data, or an honest client inadvertently issuing incorrect updates (e.g., re-adding an existing keyword-identifier pair or attempting to delete a non-existent one). In VDSSE, such errors can lead to servers returning invalid search results or clients incorrectly rejecting valid ones. To ensure reliable service operation, a practical DSSE must therefore achieve both verifiability and fault tolerance. Regarding verifiability against malicious servers, Kurosawa et al. [11] introduced a UC-secure VDSSE supporting single-keyword queries. Bost et al. [10] proposed the GSV scheme, though it lacks forward privacy. Recognizing the importance of forward privacy in this context, Zhang et al. [12] developed a forward-private VDSSE, but their construction does not provide backward privacy. More recent advancements include the works of Zhu et al. [43] and Zhao et al. [44], who proposed verifiable DSSE schemes with both forward privacy and Type-II backward privacy.

Most recently, Das et al. [45] presented a theoretical construction that aims to combine fault tolerance, verifiability, forward privacy, and backward privacy. However, their scheme achieves only Type-II backward privacy, which is a weaker notion compared to the Type-I$^-$ backward privacy targeted in our work. Additionally, as a theoretical proposal, it does not include implementation or experimental evaluation, leaving its practical performance unverified

Despite these advancements, including the recent theoretical effort by Das et al. [45], a significant gap remains for deploying comprehensively secure and reliable query services in practical cloud environments. Existing schemes often excel in specific dimensions while falling short in others, resulting in incomplete solutions that lack either strong backward privacy, fault tolerance, or verifiability. Consequently, there is a clear absence of a high-performance DSSE that simultaneously achieves fault tolerance, result verifiability, forward privacy, and strong (Type-I$^-$) backward privacy—a combination essential for real-world cloud database service deployment. Our work, FVDSSE, is the first to bridge this gap effectively, providing a comprehensively secure, reliable, and efficient solution backed by extensive experimental validation on real-world datasets.

III. System Overview and Security Objectives

Building on the analysis in Section II, existing DSSE schemes have advanced specific properties like verifiability, fault tolerance, or privacy in isolation, leaving a critical gap for practical cloud database services that require these properties simultaneously. To address this, FVDSSE is designed as an integrated solution. In this section, we establish the formal framework for FVDSSE. We begin by defining the System and Interaction Model, which captures the two-party client-server architecture of our encrypted database service. Next, we delineate the Threat Model, specifying the adversarial capabilities we defend against, including a malicious server and an operationally fallible client. We then detail the Protocol Workflow and Operations, describing the core algorithms that constitute the service. Finally, we articulate the core Design Objectives and Security Guarantees of FVDSSE, which explicitly link fault tolerance to operational resilience, verifiability to service trustworthiness, and strong privacy to long-term data confidentiality. This comprehensive formal foundation underpins the detailed construction and security evaluation of FVDSSE in the subsequent sections.

A. System and Interaction Model

Our system targets large-scale cloud environments where a resource-constrained data owner outsources an encrypted database (EDB) to a powerful cloud server modeled as malicious. Two system entities participate:

- **Data Owner**: holds the plaintext data collection, bootstraps the system, and issues all subsequent search and update tokens.
- **Cloud Server**: stores EDB, executes search or update operations upon request, and returns encrypted answers together with cryptographic proofs.

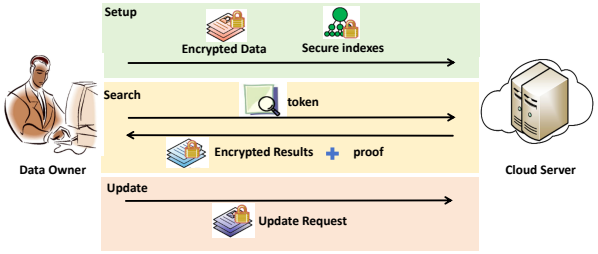


Fig. 1: System Model

As illustrated in Fig. 1, the workflow is:

1) The data owner encrypts the local database and uploads EDB to the server.
2) The data owner can dynamically update EDB by sending add or delete tokens to the server .
3) To search for a keyword, the data owner generates a search token; the server computes the matching (encrypted) identifiers and a verification proof.

Deployment. In a real-world cloud deployment, the cloud server is realized as an encrypted database proxy that exposes standard query interfaces (e.g., via gRPC) to upper-layer applications. When this proxy receives a query request, it translates the request into the corresponding FVDSSE protocol messages, interacts with the persistent backend (e.g., RocksDB) that stores the encrypted database (EDB), and returns the verified results to the client. The data owner, meanwhile, serves as a database administrator, taking on responsibilities such as system initialization, key management, and the generation of update operations. Additionally, to ensure operational resilience, the client state σ must be durably stored in a reliable configuration store, while the actual keys themselves are backed up using secure offline storage or a dedicated

backup mechanism controlled solely by the client. Overall, these design choices preserve all security properties—fault tolerance, verifiability, forward privacy, and Type-I$^-$ backward privacy—while enabling seamless integration with existing cloud infrastructure and operational best practices.

B. Threat Model

The threat model considers a malicious server that may arbitrarily deviate from the protocol—for instance, by tampering with the encrypted database or returning incorrect search results. The server observes the full encrypted database, every update and search token, and the timing and size of each interaction. Security requires that the server learns no plaintext information beyond what is formally permitted by the leakage functions $\mathcal{L}^{\text{Update}}$ and $\mathcal{L}^{\text{Search}}$. The data owner is honest but operationally fallible and may issue faulty updates such as deleting a non-existent keyword-identifier pair. The system must maintain correctness and security despite such client errors. What 's more, client storage is trusted, communication channels are authenticated, and side-channel attacks are considered out of scope.

C. Protocol Workflow and Operations

The proposed encrypted database service is realized through four well-defined operations between the data owner (acting as the client in the protocol) and the server. Each operation is specified as follows:

- **System Initialization (Setup)**
  The client takes a security parameter λ and a plaintext database $\texttt{DB}$[2] as inputs, and generates an encrypted database $\texttt{EDB}$ along with a local private state σ. σ contains cryptographic keys and data structures required for subsequent operations. The client uploads $\texttt{EDB}$ to the server and securely stores σ locally.
- **Data Update (Update)**
  The update operation is a two-phase protocol:
  1) Based on the current state σ, an operation type $op \in \{add, del\}$, and a keyword-identifier pair (ind, w), the client generates an update token ut and updates its local state to σ'.
  2) After receiving ut, the server modifies the encrypted database $\texttt{EDB}$ accordingly, producing an updated $\texttt{EDB}'$.
- **Search Query (Search)**
  The search process also consists of a client phase and a server phase:
  1) The client generates a search token st from the query keyword w and the state σ, and sends st to the server.
  2) Using st, the server performs the search over the encrypted database $\texttt{EDB}$ and returns an encrypted result set $\mathbf{R}$ together with a corresponding integrity proof π.

  The client then invokes the verification algorithm $\mathbf{Verify}$ to check the validity of the $\mathbf{R}$. If verification passes, in the result-hiding model the client decrypts the $\mathbf{R}$ to obtain the plaintext identifier list. If verification fails, the client outputs $\texttt{Reject}$.
- **Result Verification (Verify)**
  Using its local state σ, the received result set $\mathbf{R}$ and proof π, the client decides whether π constitutes a valid proof for the correctness of $\mathbf{R}$. It outputs $\texttt{Accept}$ on success and $\texttt{Reject}$ otherwise.

[2]We denote by $|D|$ the total number of files in the database $\texttt{DB}$.

D. Design Objectives and Security Guarantees

We design $\texttt{FVDSSE}$ to deliver a trustworthy encrypted database service by simultaneously guaranteeing four essential properties. Each property addresses a critical aspect of service quality in the presence of an untrusted cloud server and an operationally fallible client.

1) Verifiability: To establish trust in an untrusted cloud environment, the client must be able to cryptographically verify the integrity and completeness of all search results. Specifically, the scheme must guarantee that the client can detect with overwhelming probability if a malicious server tampers with or fabricates any part of the result set corresponding to a search query. This property, often termed verifiability, is the cornerstone of service trustworthiness, ensuring clients receive provably correct answers.

2) Strong Privacy: Updates must not reveal which previous or future searches are affected. We achieve the standard notions of forward and backward privacy.

Forward privacy hides the content of newly added entries from past queries:

Definition 1 (Forward Privacy [13])**.** *A $\mathcal{L}$-adaptively-secure DSSE scheme is forward-private iff the update leakage function $\mathcal{L}^{Update}$ can be written as*

$$\mathcal{L}^{Update}(op, in) = \mathcal{L}'(op, \{(ind_i, \mu_i)\}),$$

where $\{(ind_i, \mu_i)\}$ is the set of modified identifier/keywords pair, μ_i is the number of updated keywords corresponding to the updated file ind_i, and $\mathcal{L}'$ is stateless.

Remark. In this paper, we only update one keyword-identifier pair for each update. Then the leakage function will be

$$\mathcal{L}^{\text{Update}}(op, w, ind) = \mathcal{L}'(op, ind)\ .$$

Backward privacy hides the fact that a file was ever present once it is deleted. In [14], Bost et al. introduced three levels of backward privacy from Type-I to Type-III (from most secure to least secure). Later, Zuo et al. [18] introduced Type-I$^-$, which is somewhat stronger than Type-I. Specifically,

- Type-I$^-$: For a search query w, it leaks the files currently matching keyword w, the total number of updates, and the corresponding update times on keyword w.
- Type-I: For a search query w, it leaks the files currently matching keyword w, when they were inserted, and the total number of updates on keyword w.

To formally define Type-I$^-$[3], we need to define a new leakage function $\texttt{Time}$. For a search query w, $\texttt{Time}(w)$

[3]Our scheme achieves Type-I$^-$ backward privacy; we mainly focus on introducing Type-I$^-$ formally and refer readers to [14] for the formal definitions of Type-I to Type-III backward privacy.

reveals the timestamps t corresponding to the updates keyword w. Formally, for a sequence of update queries Q

$$\texttt{Time}(w) = \{t : (t, op, w, ind) \in Q'\} \ .$$

Definition 2 (Type-I$^-$ Backward Privacy [18]). *A adaptively-secure DSSE scheme is Type-I$^-$ backward-priv iff the update and search leakage functions $\mathcal{L}^{Update}$, $\mathcal{L}^{Sea}$ can be written as*

$$\mathcal{L}^{Update}(op, w, ind) = \mathcal{L}'(op) \, ,$$

$$\mathcal{L}^{Search}(w) = \mathcal{L}''(\texttt{sp}(w), \texttt{rp}(w), \texttt{Time}(w)) \, ,$$

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

3) Fault Tolerance: As mentioned before, a client r unconsciously issue faulty updates (add an existing keywc identifier pair or delete a non-existent keyword-identifier pa which can cause the server to return incorrect search res (i.e., semi-honest model) or the client to refuse to accept (correct) search results in VDSSE (i.e., malicious model). an update query (op, w, ind), the faulty updates are defined as follows:

1) $op = \text{add}, ind \in \texttt{DB}(w)$,
2) $op = \text{del}, ind \notin \texttt{DB}(w)$.

A fault-tolerant DSSE scheme guarantees that when these faulty updates occur in the presence of server failures, the client still obtains the correct and complete search results $\texttt{DB}(w)$ with integrity.

IV. System Design

We now present the architecture and protocols of our encrypted-search service (FVDSSE) designed specifically for cloud database environments. The design realises the four non-negotiable guarantees derived in Section III—verifiability, forward privacy, Type-I$^-$ backward privacy, and client-side fault tolerance—while using only fast symmetric-key primitives. For clarity we proceed in four modular stages: one-time system initialisation (**Setup**), on-line document maintenance (**Update**), privacy-preserving search (**Search**), and lightweight result auditing (**Verify**). A caching-enhanced variant FVDSSE−C is described afterwards; it inherits the same security properties while accelerating repetitive queries. Fig. 2 illustrates the high-level interaction flow of the four stages above, abstracting the client-server message exchanges; black arrows denote the core protocol, while red arrows highlight the caching extension introduced in FVDSSE−C.

A. FVDSSE Core Protocols

The FVDSSE service is decomposed into four tightly-coupled protocols that together realise verifiability, forward and Type-I$^-$ backward privacy, and client-side fault tolerance while using only fast symmetric-key primitives. Each protocol is stateless on the server side and requires only a small, constant-size secret state σ at the client. The subsequent four sections describe the workflows in the order they are exercised during the system life-cycle: one-time initialisation, on-line updates, privacy-preserving search, and lightweight

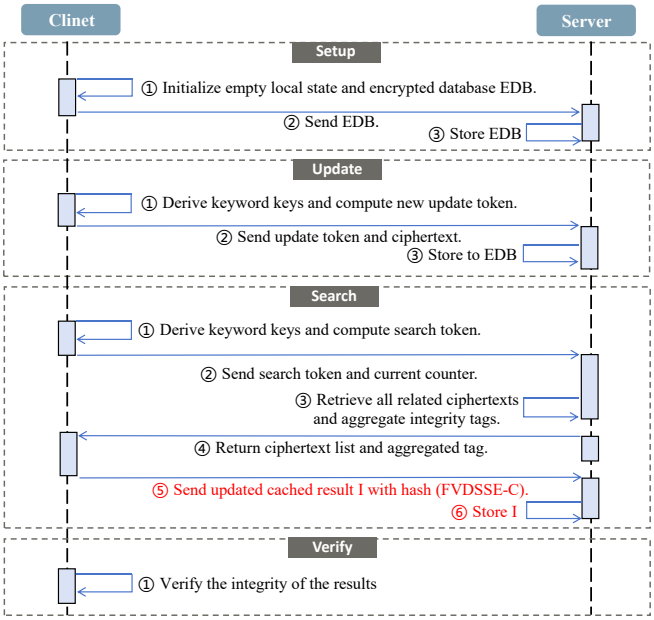


Fig. 2: High-level interaction flow of FVDSSE and FVDSSE−C

result auditing. For brevity, the full algorithmic listings are collected in Algorithm 1; the prose below focuses on the operational flow and design rationale.

1) System Initialisation: Deployment begins with the data owner executing the **Setup** algorithm (Algorithm 1). This phase transforms the local plaintext collection into an encrypted database (EDB) that can be safely outsourced to the cloud, while retaining a compact private state σ for all subsequent operations. The owner keeps σ on a trusted device; the server receives only EDB.

2) Secure Update Protocol: Dynamic environments require frequent add/delete operations. Our secure **Update** algorithm (Algorithm 1) allows the owner to insert or remove a keyword-identifier pair (w, ind) without revealing the keyword to the server and without leaking linkability across epochs. Forward privacy is enforced via the "symmetric one-way function" technique: each new update is tied to a fresh search token ST_{c+1} while the previous token ST_c is hidden through the keyed hash H_2. Type-I$^-$ backward privacy is achieved by encrypting the operation together with the pair (w, ind) under an IND-CPA secure symmetric cipher, ensuring that deletions leave no exploitable trace. Finally, a keyed hash H_3 over the ciphertext provides a client-verifiable integrity tag that will later be used for verification.

3) Privacy-Preserving Search Protocol: When an authorised user wishes to retrieve all files matching keyword w, the client runs the **Search** algorithm (Algorithm 1). The client first retrieves the current counter c for w from its local map **CT**, derives the keyword-specific keys K_w, K'_w, and transmits the search token ST_c to the server. The server uses ST_c to locate every ciphertext associated with w, combines the embedded integrity tags via XOR, and returns the ciphertext list **R** together with the aggregated tag h. This aggregation reduces both bandwidth and client-side computation to a single hash comparison.

Algorithm 1 FVDSSE

Setup(1^λ)

$K_s, K_t \xleftarrow{\$} \{0,1\}^\lambda$, **CT**, EDB $\leftarrow$ empty map
return $(\sigma = (K_s, K_t, \mathbf{CT}), \texttt{EDB})$

Update$(op, w, ind, \sigma; \texttt{EDB})$

Client:

$K_w||K'_w \leftarrow F_{K_s}(w)$, $c \leftarrow \mathbf{CT}[w]$
if $c = \perp$ **then**
  $c \leftarrow -1$
end if
$ST_c \leftarrow F_{K_t}(w||c)$
$ST_{c+1} \leftarrow F_{K_t}(w||c+1)$
$\mathbf{CT}[w] \leftarrow c+1$
$UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
$C_{ST_c} \leftarrow H_2(K_w, ST_{c+1}) \oplus ST_c$
$e_{c+1} \leftarrow \texttt{SE.Enc}(K'_w, op||ind||c+1)$
$h_{c+1} \leftarrow H_3(K'_w, e_{c+1})$
$he_{c+1} \leftarrow H_4(K_w, ST_{c+1}) \oplus e_{c+1}$
$hh_{c+1} \leftarrow H_5(K_w, ST_{c+1}) \oplus h_{c+1}$
Send $(UT_{c+1}, (he_{c+1}, C_{ST_c}, hh_{c+1}))$ to server.

Server:

$\texttt{EDB}[UT_{c+1}] \leftarrow (he_{c+1}, C_{ST_c}, hh_{c+1})$

Search$(w, \sigma; \texttt{EDB})$

Client:

$K_w||K'_w \leftarrow F_{K_s}(w)$, $c \leftarrow \mathbf{CT}[w]$
if $c = \perp$ **then**
  return $\perp$
end if
$ST_c \leftarrow F_{K_t}(w||c)$
Send (K_w, ST_c, c) to server.

Server:

$\mathbf{R} \leftarrow$ empty set, $h \leftarrow 0$
for $i \leftarrow c$ to 0 **do**
  $UT_i \leftarrow H_1(K_w, ST_i)$
  $(he_i, C_{ST_{i-1}}, hh_i) \leftarrow \texttt{EDB}[UT_i]$
  $e_i \leftarrow H_4(K_w, ST_i) \oplus he_i$, $h_i \leftarrow H_5(K_w, ST_i) \oplus hh_i$
  $\mathbf{R} \leftarrow \mathbf{R} \cup e_i$, $h \leftarrow h \oplus h_i$
  $ST_{i-1} \leftarrow H_2(K_w, ST_i) \oplus C_{ST_{i-1}}$
end for
Send $(\mathbf{R}, h)$ to the client.

Client:

if Verify$(K'_w, c, \mathbf{R}, h) = \texttt{Reject}$ **then**
  return Reject
end if
$\mathbf{R}_d, \mathbf{I} \leftarrow$ empty set
for $e_i \in \mathbf{R}$ **do**
  $op||ind||i \leftarrow \texttt{SE.Dec}(K'_w, e_i)$, $\mathbf{R}_d \leftarrow \mathbf{R}_d \cup (op||ind||i)$
end for
for $i = 0$ to c **do**
  Get $(op||ind||i)$ from $\mathbf{R}_d$
  if $op = add$ **then**
    $\mathbf{I} \leftarrow \mathbf{I} \cup ind$
  else
    $\mathbf{I} \leftarrow \mathbf{I} \setminus ind$
  end if
end for
return $\mathbf{I}$

Verify$(K'_w, c, \mathbf{R}, h)$

if $|\mathbf{R}| \neq c$ **then**
  return Reject
end if
$h' \leftarrow 0$
for $e_i \in \mathbf{R}$ **do**
  $h_i \leftarrow H_3(K'_w, e_i)$, $h' \leftarrow h' \oplus h_i$
end for
if $h' = h$ **then**
  return Accept
else
  return Reject
end if

4) Result-Verification Protocol: Integrity is enforced by the **Verify** algorithm (Algorithm 1). The client first checks that the returned list length equals the expected counter value c; if not, it outputs Reject. Otherwise it recomputes the XOR of the keyed hashes (H_3) over the ciphertexts and compares the result with the aggregated tag h provided by the server. A match implies that **(i)** every expected update is present, **(ii)** no spurious update was injected, and **(iii)** the server did not suppress legitimate entries—thus the client either decrypts the identifiers or issues Reject.

B. Accelerated Variant: FVDSSE-C

Since the client decrypts the search results and will retrieve the actual files, the search results (final identifiers) are revealed to the server. Then we can let the server store the results to reduce the computation time for the (same) future queries. In addition, the revealed search results can be saved in continuous physical addresses to improve locality [16]. To achieve this, we additionally add a new map (named $\texttt{EDB}_c$), which is used to store the search results of previously searched queries. In addition, we need two counters (c_1, c_2), where c_1 is used to denote the number of searches for each searched queries and c_2 denotes the number of new updates (similar to the c in FVDSSE). The new scheme (named FVDSSE−C) is described in Algorithm 2, and the details are as follows:

The extension from FVDSSE to FVDSSE−C introduces a caching mechanism for better performance on repetitive queries. The **Setup** algorithm is almost identical to that of FVDSSE, with the key difference being that the client additionally initializes a new cache map $\texttt{EDB}_c$ and sends it (together with $\texttt{EDB}_a$) to the server.

Building on this, the **Update** algorithm is similar but now uses two counters (c_1, c_2) instead of one (c). Here, c_1 denotes the number of searches for a keyword, while c_2 tracks the number of updates since the last search. Consequently, the keyword keys K_w and K'_w are derived from both the keyword w and the search counter c_1, and the search token is similarly linked to both c_1 and c_2.

This modification enables an optimized **Search** algorithm. While similar to the base scheme, the client now additionally generates a search token old_w for previously cached results and sends it to the server. This allows the server to retrieve all relevant search results from both the cache and the main database. Finally, the client sends the consolidated results back to the server to refresh the cache for future queries.

The **Verify** algorithm is also extended. Apart from verifying the new search results **R** as in FVDSSE, it additionally takes the previously cached results $\mathbf{I}_o$ and their corresponding hash $h_{\mathbf{I}_o}$ as input to validate their integrity.

Algorithm 2 FVDSSE−C, differences are in red

Setup(1^λ)

$K_s, K_t \xleftarrow{\$} \{0,1\}^\lambda$, **CT**, $\mathtt{EDB}_a, \mathtt{EDB}_c \leftarrow$ empty map
return $(\sigma = (K_s, K_t, \mathbf{CT}), \mathtt{EDB} = (\mathtt{EDB}_a, \mathtt{EDB}_c))$

Update$(op, w, ind, \sigma; \mathtt{EDB})$

Client:

$(c_1, c_2) \leftarrow \mathbf{CT}[w]$
if $c_2 = \perp$ **then**
  if $c_1 = \perp$ **then**
    $c_1 \leftarrow 0$
  end if
  $c_2 \leftarrow -1$
end if
$ST_c \leftarrow F_{K_t}(w||c_1||c_2)$
$K_w||K'_w \leftarrow F_{K_s}(w||c_1)$
$ST_{c+1} \leftarrow F_{K_t}(w||c_1||c_2+1)$
$\mathbf{CT}[w] \leftarrow (c_1, c_2+1)$
$UT_{c+1} \leftarrow H_1(K_w, ST_{c+1})$
$C_{ST_c} \leftarrow H_2(K_w, ST_{c+1}) \oplus ST_c$
$e_{c+1} \leftarrow \mathtt{SE.Enc}(K'_w, op||ind||c_2+1)$
$h_{c+1} \leftarrow H_3(K'_w, e_{c+1})$
$he_{c+1} \leftarrow H_4(K_w, ST_{c+1}) \oplus e_{c+1}$
$hh_{c+1} \leftarrow H_5(K_w, ST_{c+1}) \oplus h_{c+1}$
Send $(UT_{c+1}, (he_{c+1}, C_{ST_c}, hh_{c+1}))$ to server.

Server:

$\mathtt{EDB}_c[UT_{c+1}] \leftarrow (he_{c+1}, C_{ST_c}, hh_{c+1})$

Search$(w, \sigma; \mathtt{EDB})$

Client:

$(c_1, c_2) \leftarrow \mathbf{CT}[w]$
if $c_1 = \perp$ **then**
  return $\perp$
end if
$K_w||K'_w \leftarrow F_{K_s}(w||c_1)$
$old_w \leftarrow \perp, ST_c \leftarrow \perp$
if $c_1 > 0$ **then**
  $old_w \leftarrow F_{K_t}(w||c_1)$
end if
if $c_2 \neq \perp$ **then**
  $ST_c \leftarrow F_{K_t}(w||c_1||c_2)$
end if
$\mathbf{CT}[w] \leftarrow (c_1+1, \perp)$
Send (K_w, old_w, ST_c, c_2) to server.

Server:

if $old_w \neq \perp$ **then**
  $(\mathbf{I}_o, h_{\mathbf{I}_o}) \leftarrow \mathtt{EDB}_c[old_w]$
  if $c_2 \neq \perp$ **then**
    Clear $\mathtt{EDB}_c[old_w]$
  end if
end if
if $c_2 \neq \perp$ **then**
  $\mathbf{R} \leftarrow$ empty set, $h \leftarrow 0$
  for $i \leftarrow c_2$ to 0 **do**
    $UT_i \leftarrow H_1(K_w, ST_i)$
    $(he_i, C_{ST_{i-1}}, hh_i) \leftarrow \mathtt{EDB}_a[UT_i]$
    Clear $\mathtt{EDB}_a[UT_i]$, $e_i \leftarrow H_4(K_w, ST_i) \oplus he_i$
    $h_i \leftarrow H_5(K_w, ST_i) \oplus hh_i$
    $\mathbf{R} \leftarrow \mathbf{R} \cup e_i$, $h \leftarrow h \oplus h_i$
    $ST_{i-1} \leftarrow H_2(K_w, ST_i) \oplus C_{ST_{i-1}}$
  end for
end if
Send $(\mathbf{R}, h, \mathbf{I}_o, h_{\mathbf{I}_o})$ to the client.

Client:

if Verify$(K'_w, c_2, \mathbf{R}, h, \mathbf{I}_o, h_{\mathbf{I}_o}) = \mathtt{Reject}$ **then**
  return Reject
end if
$\mathbf{R}_d, \mathbf{I} \leftarrow$ empty set, $\mathbf{I} \leftarrow \mathbf{I} \cup \mathbf{I}_o$
for $e_i \in \mathbf{R}$ **do**
  $op||ind||i \leftarrow \mathtt{SE.Dec}(K'_w, e_i)$, $\mathbf{R}_d \leftarrow \mathbf{R}_d \cup (op||ind||i)$
end for
for $i = 0$ to c_2 **do**
  Get $(op||ind||i)$ from $\mathbf{R}_d$
  if $op = add$ **then**
    $\mathbf{I} \leftarrow \mathbf{I} \cup ind$
  else
    $\mathbf{I} \leftarrow \mathbf{I} \setminus ind$
  end if
end for
Client outputs $\mathbf{I}$
if $c_2 \neq \perp$ **then**
  $old_w \leftarrow F_{K_t}(w||c_1+1)$, $h_{\mathbf{I}} \leftarrow H_3(K'_w, \mathbf{I})$
  Send $(old_w, \mathbf{I}, h_{\mathbf{I}})$ to sever.
end if

Server:

$\mathtt{EDB}_c[old_w] \leftarrow (\mathbf{I}, h_{\mathbf{I}})$

Verify$(K'_w, c_2, \mathbf{R}, h, \mathbf{I}_o, h_{\mathbf{I}_o})$

if $|\mathbf{R}| \neq c_2$ **then**
  return Reject
end if
if $(\mathbf{I}_o, h_{\mathbf{I}_o}) \neq \perp$ **then**
  $h_o \leftarrow H_3(K'_w, \mathbf{I}_o)$
  if $h_o \neq h_{\mathbf{I}_o}$ **then**
    return Reject
  end if
end if
$h' \leftarrow 0$
for $e_i \in \mathbf{R}$ **do**
  $h_i \leftarrow H_3(K'_w, e_i)$, $h' \leftarrow h' \oplus h_i$
end for
if $h' = h$ **then**
  return Accept
else
  return Reject
end if

C. Security Analysis

In this section, we mainly analyze the *fault-tolerance, verifiability, confidentiality* of FVDSSE. For FVDSSE−C, it directly inherits the *fault-tolerance, verifiability, confidentiality* of FVDSSE. This is due to the fact that FVDSSE−C only caches the already leaked search results, which does not incur new leakages.

Theorem 1. *(Fault-tolerance of FVDSSE) FVDSSE is fault-tolerant.*

Proof. The fault-tolerance of FVDSSE is quite straightforward. To avoid the repeated updates (e.g., add an existing keyword-identifier pair) to be canceled out in the proof π (verifiability), the keyword-identifier pair is encrypted together with the counter c to avoid repetition. Then the repeated updates will correspond to different hash value, where they cannot be canceled out by the combine operation (i.e., xor) in the proof π. In addition, the server cannot trick the client into accepting incorrect search results. As a result, the client will always accept the correct search results returned by the server. To get the final results, the client sequentially filter all the search results returned by the server. If first fault update happens (i.e., add an existing keyword-identifier pair), only one identifier ind remains in $\mathbf{I}$ according to the *set union*

operation. Similarly, if the second fault update happens (i.e., delete a non-existed keyword/identifier), no identifier will be removed from $\mathbf{I}$ according to the *set difference* operation. As a result, the client will get the correct final results. □

The verifiability of FVDSSE relies on the collision resistance of the keyed hash function H_3. Without the secret key K'_w, the server cannot forge a valid hash value. Formally,

Theorem 2. *(Verifiability of FVDSSE) Let H_3 be a collision-resistant keyed hash function. Then FVDSSE is verifiable.*

Proof. Consider the interaction between an honest client (the challenger $\mathcal{C}$) and a malicious server (the adversary $\mathcal{A}$). $\mathcal{C}$ first initializes the system via $\mathbf{Setup}$. $\mathcal{A}$ may then adaptively issue search queries. For a target keyword w, $\mathcal{A}$ receives the necessary search token from $\mathcal{C}$. To break verifiability, $\mathcal{A}$ must respond with a pair $(\mathbf{R}, h)$ that passes the client's verification, even though $\mathbf{R}$ is not the correct encrypted result set for w. The verification algorithm of FVDSSE computes a check value from $\mathbf{R}$ using the secret K'_w and compares it with the received h. Thus, for $\mathcal{A}$ to succeed, it must forge a valid h for an incorrect $\mathbf{R}$ without knowledge of K'_w. The probability of such a forgery is bounded by the security of the underlying cryptographic hash function H_3. Since H_3 is collision-resistant (or pseudorandom), any probabilistic polynomial-time adversary $\mathcal{A}$ can succeed in this forgery only with negligible probability. Therefore, the scheme ensures verifiability: the client will accept an incorrect result with at most negligible probability. □

The confidentiality (forward and Type-I$^-$ backward privacy) of FVDSSE relies on the secure PRF, IND-CPA secure symmetric encryption, and random oracles. Formally

Theorem 3. *(Forward and Type-I$^-$ backward privacy of FVDSSE) Let F be a secure PRF, $SE = (SE.\mathtt{Setup}, SE.\mathtt{Enc}, SE.\mathtt{Dec})$ be a IND-CPA secure symmetric encryption, and H_1, H_2, H_4, H_5 be random oracles and output λ bits. We define $\mathcal{L}_{FVDSSE} = (\mathcal{L}^{Search}_{FVDSSE}, \mathcal{L}^{Update}_{FVDSSE})$, where $\mathcal{L}^{Search}_{FVDSSE}(w) = (sp(w), rp(w), \mathtt{Time}(w))$ and $\mathcal{L}^{Update}_{FVDSSE}(op, w, ind) = \perp$. Then FVDSSE is $\mathcal{L}_{FVDSSE}$-forward and Type-I$^-$ backward private.*

Proof. FVDSSE ensures forward privacy by updating only one keyword-identifier pair per operation and implementing a symmetric one-way function technique: for each new update, the client generates a random search token ST_{c+1} via the keyed hash function H_2 to hide the previous token ST_c, ensuring the server cannot associate new files with previous queries. For Type-I$^-$ backward privacy, the client encrypts each operation and keyword-identifier pair using a CPA secure symmetric encryption scheme, restricting leakage to current matching files, total update count, and update timestamps for queried keywords. This aligns with the stateless leakage functions defined for Type-I$^-$ backward privacy. In summary, the forward privacy of FVDSSE is supported by the collision resistance of the hash function, while Type-I$^-$ backward privacy depends on the CPA security of the encryption scheme. □

Remark. The leakages of FVDSSE−C are almost the same as the leakages of FVDSSE except that FVDSSE−C caches the already leaked search results for future queries, which does not incur new leakages. Hence, it directly inherits the *fault-tolerance, verifiability, and confidentiality* of FVDSSE.

The caching mechanism in FVDSSE−C reveals only that a particular keyword has been searched before, which is already leaked by the search pattern in the base FVDSSE scheme. Since the cached results consist solely of search results already disclosed to the server in prior searches, no additional information about the encrypted database is exposed beyond the leakage functions defined in Section III-D. Hence, FVDSSE−C retains all security properties of FVDSSE.

V. Experimental Evaluation

We benchmark the FVDSSE family as a production-grade encrypted-search service running on commodity cloud nodes. The goal is to quantify the end-to-end latency, throughput, storage cost, and bandwidth overhead that a cloud provider would observe when offering verifiable, forward and backward privacy and fault-tolerant search on million-document datasets.

A. Evaluation Objectives and Setup

Objectives. (1) Confirm that the integrated security stack (verifiability, strong privacy, fault-tolerance) remains practical at cloud scale. (2) Measure protocol-level latency (search or update) and resource cost against the strongest partial-security baseline available. (3) Demonstrate horizontal scalability—how latency grows when the encrypted corpus scales from 1 M to 65 M keyword-identifier pairs.

Test-bed. Two geo-distributed VMs emulate a realistic cloud edge-core deployment:

- Client VM: 24-core Intel Xeon Silver 4310 @ 2.10 GHz, 54 GB RAM, 491 GB SSD, Ubuntu 18.04.
- Server VM: 16-core Intel Core (Broadwell) @ 2.15 GHz, 64 GB RAM, 4 TB HDD, Ubuntu 18.04.

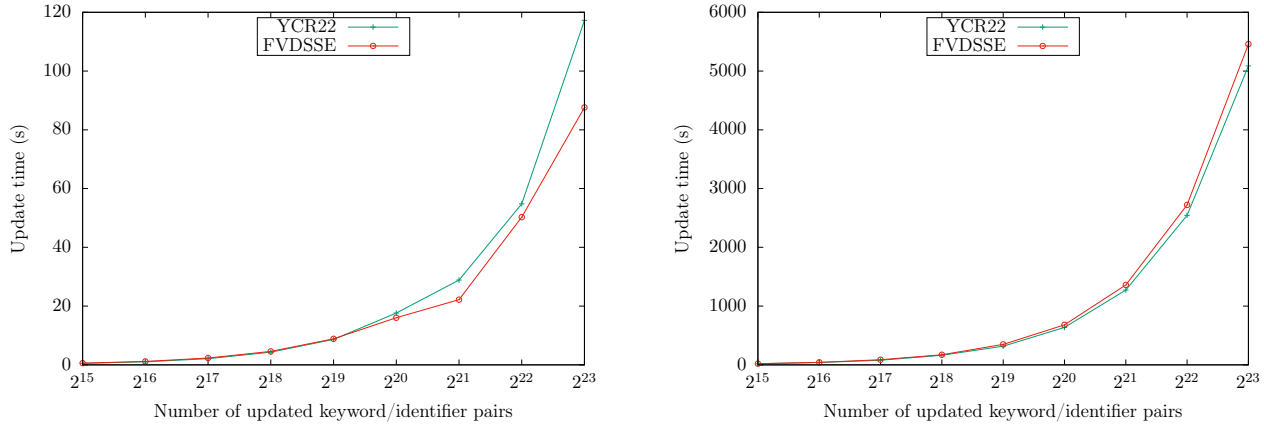
RTT$\approx 85\,\mathrm{ms}$(WAN);0.02 ms(local loop-back).Code:C++17, Crypto++ (AES-CTR-128, SHA-256) [46], RocksDB [47] (persistent index), gRPC [48] (TLS 1.3 channel). Source available at: https://anonymous.4open.science/r/FVDSSE-S-AE07.

Dataset. We use the real-world corpus YCR22−C containing 4.7×10^5 keywords, 1.1×10^6 documents, 8.4×10^6 keyword-identifier pairs, plus four synthetic expansions (1M, 4M, 16M, 65M pairs) to stress-test scalability.

Baselines. YCR22 and YCR22−C are the state-of-the-art fault-tolerant DSSE systems; both provide weaker privacy (no backward privacy) and heavier crypto (authenticated encryption). ROSE is excluded because its public-key core is orders of magnitude slower YCR22−C .

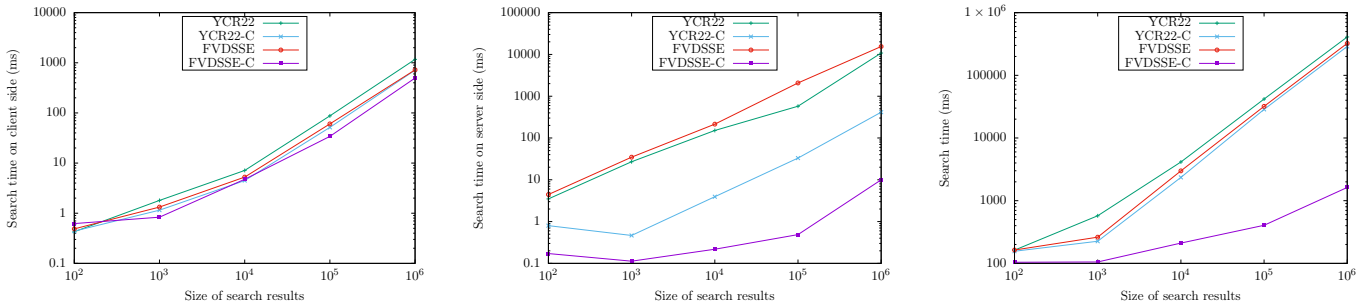
B. Storage Overhead Analysis

To test the server storage, we set up the encrypted database for YCR22 and FVDSSE with four synthetic datasets. After the setup, we measure the size of the RocksDB database on the server side. The results are shown in Table II. From the table, we can see that the server storage required by FVDSSE



(a) Local setting (b) WAN setting

Fig. 3: **Performance of the update query**



(a) Client side (b) Server side (c) WAN setting

Fig. 4: **Performance of the search query**

TABLE II: Server storage overhead

Scheme	Number of keyword-identifier pairs			
	1×10^6	4×10^6	1.6×10^7	6.5×10^7
YCR22	179MB	488MB	1.7GB	6.2GB
FVDSSE	182MB	503MB	1.8GB	7.0GB

TABLE III: Communication cost in batch updates

Scheme	Number of keyword-identifier pairs								
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
YCR22	3.1MB	6.2MB	13MB	25MB	49MB	98MB	196MB	392MB	783MB
FVDSSE	3.3MB	6.5MB	13MB	26MB	52MB	104MB	208MB	416MB	831MB

TABLE IV: Communication cost of the search query

Scheme	Size of search results				
	10^2	10^3	10^4	10^5	10^6
YCR22	8KB	48KB	480KB	4.7MB	47MB
YCR22-C	4KB	28KB	248KB	2.7MB	27MB
FVDSSE	4KB	16KB	148KB	1.5MB	15MB
FVDSSE-C	4KB	8KB	72KB	684KB	6.7MB

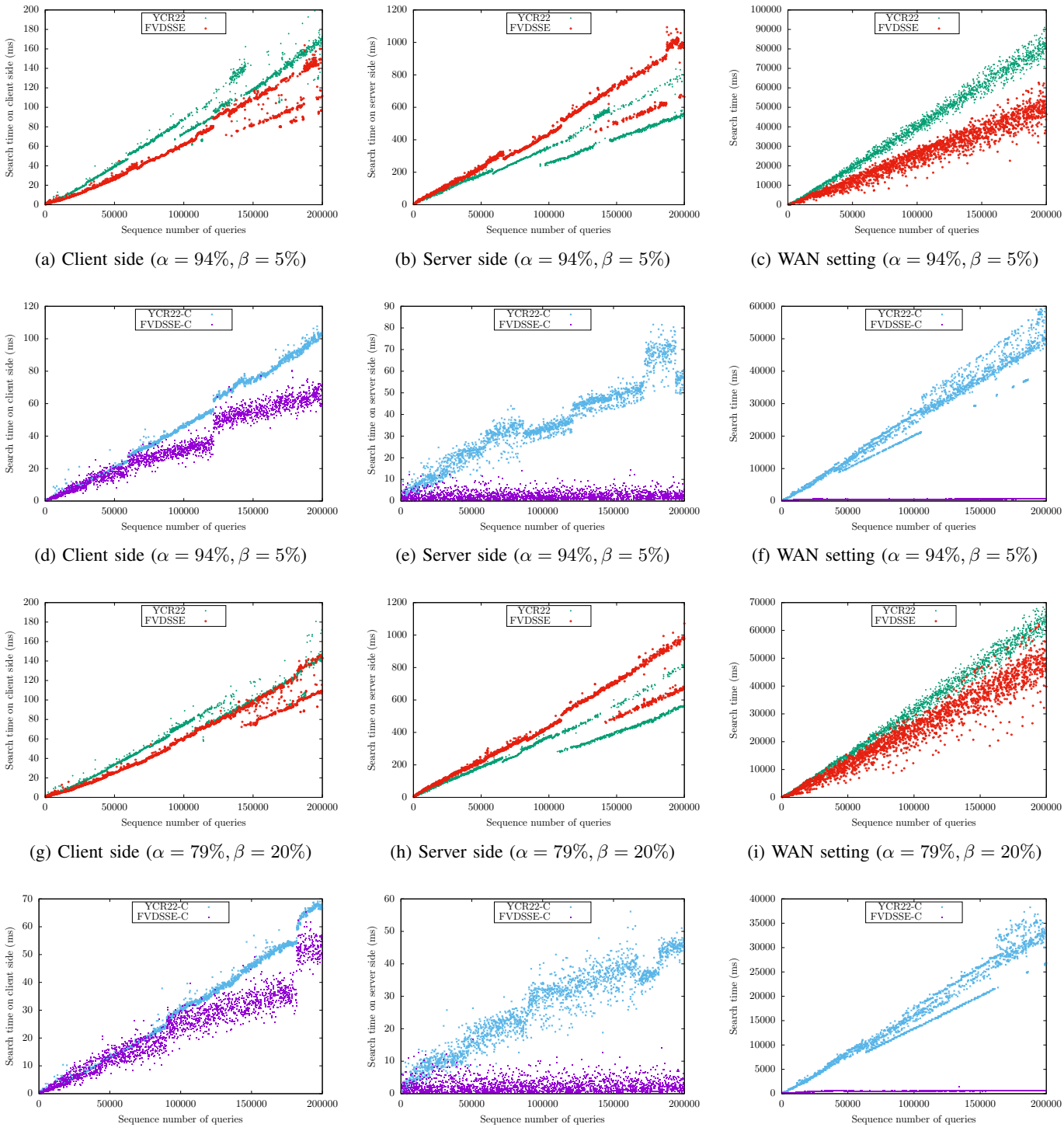
is comparable to that required by YCR22. Note that, for the dataset of the same size, the maximum value of the server side storage required by YCR22-C and FVDSSE-C is the same as the server storage for YCR22 and FVDSSE, respectively.

C. Dynamic Operation Performance

1) Update Latency and Bandwidth: To comprehensively measure the update performance of YCR22 and FVDSSE[4], we insert the different number of keyword-identifier pairs into the database in the local and WAN setting, respectively. Fig. 3a and Fig. 3b present the update time in the local and WAN setting, respectively.

Fig. 3a demonstrates that the update efficiency of FVDSSE is better than YCR22 when the data scale exceeds 2^{19} keyword/identifier pairs. This advantage stems from two key factors: First, FVDSSE employs lightweight cryptographic primitives such as symmetric hash functions and AES-CTR, avoiding slow authenticated encryption of YCR22. Second, the update logic of YCR22 relies on historical states, such as computing the oldest value, introducing non-negligible overheads in memory access as data scale increases. In contrast, the stateless hash design in FVDSSE avoids such state dependencies, maintaining a smaller increase. Empirical

[4]Note that we do not specifically test the update efficiency of YCR22-C and FVDSSE-C because their update efficiency is the same as that of YCR22 and FVDSSE, respectively.



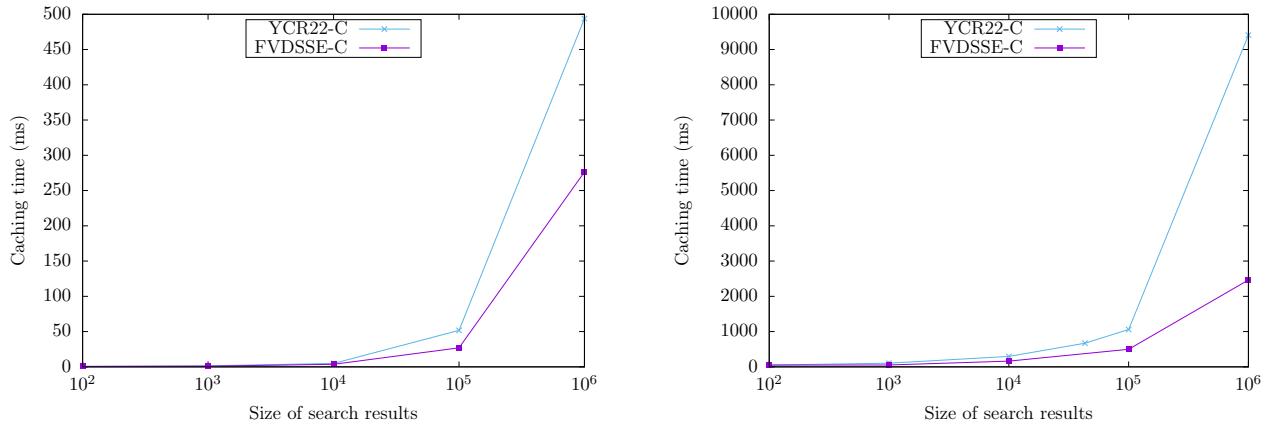
(a) Client side ($\alpha = 94\%, \beta = 5\%$) (b) Server side ($\alpha = 94\%, \beta = 5\%$) (c) WAN setting ($\alpha = 94\%, \beta = 5\%$)

(d) Client side ($\alpha = 94\%, \beta = 5\%$) (e) Server side ($\alpha = 94\%, \beta = 5\%$) (f) WAN setting ($\alpha = 94\%, \beta = 5\%$)

(g) Client side ($\alpha = 79\%, \beta = 20\%$) (h) Server side ($\alpha = 79\%, \beta = 20\%$) (i) WAN setting ($\alpha = 79\%, \beta = 20\%$)

(j) Client side ($\alpha = 79\%, \beta = 20\%$) (k) Server side ($\alpha = 79\%, \beta = 20\%$) (l) WAN setting ($\alpha = 79\%, \beta = 20\%$)

Fig. 5: **Performance of search queries in the trace simulation**



(a) Local setting (b) WAN setting

Fig. 6: **Performance of caching search results**

results validate this gap: when the data size is 2^{21}, FVDSSE is 20% faster, but it increases to 25% at 2^{23}.

In the WAN setting, the update efficiency of FVDSSE is slightly weaker than the update efficiency of YCR22, because FVDSSE has a bit more communication overhead than YCR22, where the communication cost is shown in Table III. Note that FVDSSE has more communication overhead because we set the length of the counter c (line 10 of the **Update** in Algorithm 1) to a fixed number (e.g., 7 bytes), which prevents the information leakage of the counter.

2) Search Latency and Communication Cost: We use the real-world dataset to extensively test the search performance of FVDSSE, FVDSSE−C, YCR22, and YCR22−C. In this test, we query five keywords that match 10^2, 10^3, 10^4, 10^5, and 10^6 documents, respectively. For each keyword, we perform the search query on the keyword 100 times and take the average as the final result. Search queries are performed in the local setting and the WAN setting. In the local setting, we obtain the time used by the client and the server for the computation, which are shown in Fig. 4a and Fig. 4b, respectively. In the WAN setting, we record the total time that includes the network latency, and the results are shown in Fig. 4c.

Computational time From Fig. 4a and Fig. 4b, we can see that FVDSSE causes significantly less computational overhead on the client side and has moderately more computational cost on the server side than YCR22. This is because FVDSSE assigns some of the verifying operations to the server side, while these operations are all done by the client side in YCR22. The computational time of FVDSSE−C consistently outperforms the computational time of YCR22−C on each side. This is due to the fact that the already searched results (cache) in FVDSSE−C are verified by a lightweight hash function, while YCR22−C needs to use an AE decryption to verify the results. In addition, the schemes from [20] need to recover the search results on the server side, while our schemes do not.

Total search time Fig. 4c shows that the search time of FVDSSE is lesser than the search time of YCR22 in the WAN setting. On average, compared with YCR22, FVDSSE gains 27% search performance improvement for each query. The performance of FVDSSE−C over YCR22−C is widen due to the low communication cost of our schemes. FVDSSE−C is 130% faster than YCR22−C on average for each search query. Table IV presents the amount of the transferred data between the client and the server for a search query. Compared with YCR22, we can see that FVDSSE saves 210% communication cost and FVDSSE−C takes 303% lighter communication overhead than YCR22−C.

D. System Performance under Realistic Workloads

1) Trace-Driven Evaluation: In Section V-C2, we test the search performance after setting up the encrypted database. However, in real-world applications, the database is frequently updated and a search is usually performed over the dynamic database. Moreover, as shown in Table I, the search efficiency of our schemes are affected by the number of updates occurred to the search keyword. In this section, to evaluate the search efficiency more accurately, we test the search performance in the dynamic setting, where we create two traces of 200,000 queries for a keyword. In a trace, each query has α probability of being an addition query, β chance of being a deletion, and 1% probability of being a search query. For the first trace, we set α to 94% and β to 5%. In the second trace, α is set to 79% and β is set to 20%. We measure the search time in the two traces, and the result is shown in Fig. 5. Search queries are performed in the local setting and the WAN setting, respectively. We record the time spent by the client and the server for the computation in the local setting, and the total time costed in the WAN setting. To clearly show the trend of the search time for each scheme, we plot the results for the schemes without and with cache separately.

Search without cache Fig. 5a-5c and Fig. 5g-5i present the search performance comparison between YCR22 and FVDSSE. From the figures, FVDSSE outperforms YCR22 for the computational time on the client side and the total search time in the WAN setting, while FVDSSE is slightly weaker than YCR22 in terms of computation efficiency on the server side. This result is consistent with the one shown in Section V-C2, as demonstrated in Fig. 4a: when the search result size reaches approximately 10^5, FVDSSE exhibits a search time close to 100 ms, whereas YCR22 shows a search time around 170 ms. This results aligns with the experimental

results presented in Fig. 5a, validating the consistency across static and dynamic evaluation scenarios. For the computational efficiency on the client side, FVDSSE outperforms YCR22 by 29% and 16% in the two traces, respectively. In the WAN setting, FVDSSE is 63% and 37% faster than YCR22 in the two traces, respectively.

Search with cache In Fig. 5d-5f and Fig. 5j-5l, we present the search time costed by FVDSSE−C and YCR22−C, and we can find that FVDSSE−C outperforms YCR22−C in every aspect. Specifically, compared with YCR22−C, for the two traces, FVDSSE−C gains 41% and 26% performance improvement for the computational overhead on the client side, respectively. FVDSSE−C improves the search overhead on the server side by 16.5× and 11.5× over YCR22−C for the two traces, respectively. Moreover, the consistent search time of FVDSSE−C in Fig. 5e and Fig. 5k originates from its EDB_c mechanism. For repeated search queries, the server leverages pre-stored EDB_c to directly retrieve previously searched results, eliminating the need for recomputation from the encrypted database. And the verification process is based on lightweight hash functions, which incur minimal computational overhead. In contrast, YCR22−C depends on time-consuming AE decryption for verification, causing its search time to increase with data scale. For the overall search efficiency in the WAN setting, FVDSSE−C achieves 44× and 30× improvement in the two traces, respectively. Compare with the first trace, we also can notice that the advantage of our schemes over the schemes from [20] is slightly decreased. This is because the higher deletion frequency in the second trace makes the size of the search result in the schemes from [20] becomes smaller, which slightly affects the search efficiency.

2) Caching Time Overhead: As mentioned before, FVDSSE−C and YCR22−C cache the search results on the server side after each search query. Meanwhile, the schemes attach a new proof to the cached search results. We test the time costed by this operation in both the local and WAN setting, the results are shown in Fig. 6. From Fig. 6, we can clearly see that the caching time spent by FVDSSE−C is much less than the one in YCR22−C, especially when the size of the search results is large. This is because FVDSSE−C produces significantly smaller proofs than the one in YCR22 and the hash operation used in FVDSSE−C is faster than the authenticated encryption used in YCR22−C.

VI. Conclusion

Although the security (forward and backward privacy) of DSSE has been extensively studied, this is not the case for fault-tolerance, which deals with the faulty updates that are unconsciously issued by the client—a critical concern in large-scale data sharing and multi-user cloud services. Until recently, Yuan et al. [20] introduced a fault-tolerant verifiable forward private DSSE, and Xu et al. [19] presented a robust (fault-tolerant) forward and backward private DSSE. While Das et al. [45] proposed a theoretical construction aiming for these combined properties, their scheme lacks a practical implementation and efficiency validation. To bridge this gap between theory and practice for cloud-based big data applications, we introduced the first fault-tolerant verifiable forward and backward private DSSE scheme(named FVDSSE). In addition, to further improve the efficiency of FVDSSE, we cache the already searched results and present the FVDSSE−C. The security analysis and experimental evaluation demonstrate the security and practicality of our proposed schemes. Compared with the state-of-the-art YCR22−C, the experiments show that FVDSSE−C gains a 130× improvement in search efficiency and saves both the communication overhead by 3×. In the future, we would like to make our schemes support more expressive queries.

References

[1] Google Drive, https://www.google.com/drive/.

[2] OneDrive, https://www.microsoft.com/en-sg/microsoft-365/onedrive/online-cloud-storage.

[3] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," *Proc. VLDB Endow.*, vol. 6, no. 5, pp. 289–300, Mar. 2013. [Online]. Available: https://doi.org/10.14778/2535573.2488336

[4] R. Poddar, T. Boelter, and R. A. Popa, "Arx: an encrypted database using semantically secure encryption," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1664–1678, Jul. 2019. [Online]. Available: https://doi.org/10.14778/3342263.3342641

[5] H. Yin, Y. Li, H. Deng, W. Zhang, Z. Qin, and K. Li, "Practical and dynamic attribute-based keyword search supporting numeric comparisons over encrypted cloud data," *IEEE Transactions on Services Computing*, vol. 16, no. 4, pp. 2855–2867, 2023.

[6] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *CCS 2012*. ACM, 2012, pp. 965–976.

[7] G. Zu, Y. Lu, and J. Li, "Comment on an attribute-based searchable encryption scheme with receiver anonymity," *IEEE Transactions on Services Computing*, vol. 17, no. 4, pp. 1875–1876, 2024.

[8] Y. Yang, Y. Hu, R. Li, X. Dong, Z. Cao, J. Shen, and S. Dou, "Lse: Efficient symmetric searchable encryption based on labeled psi," *IEEE Transactions on Services Computing*, vol. 17, no. 2, pp. 563–574, 2024.

[9] H. Wang, J. Ning, W. Wu, C. Lin, and K. Zhang, "Ka^{2}2se: Key-aggregation authorized searchable encryption scheme for data sharing in wireless sensor networks," *IEEE Transactions on Services Computing*, vol. 18, no. 1, pp. 226–238, 2025.

[10] R. Bost, P. Fouque, and D. Pointcheval, "Verifiable dynamic symmetric searchable encryption: Optimality and forward security," *ePrint*, vol. 2016, p. 62, 2016. [Online]. Available: http://eprint.iacr.org/2016/062

[11] K. Kurosawa, K. Sasaki, K. Ohta, and K. Yoneyama, "Uc-secure dynamic searchable symmetric encryption scheme," in *IWSEC 2016*. Springer, 2016, pp. 73–90.

[12] Z. Zhang, J. Wang, Y. Wang, Y. Su, and X. Chen, "Towards efficient verifiable forward secure searchable symmetric encryption," in *ESORICS 2019*, vol. 11736. Springer, 2019, pp. 304–321.

[13] R. Bost, "$\sigma o \phi o \varsigma$: Forward secure searchable encryption," in *CCS 2016*. ACM, 2016, pp. 1143–1154.

[14] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *CCS 2017*. ACM, 2017, pp. 1465–1482.

[15] Verizon, https://www.verizon.com/business/resources/reports/dbir/.

[16] D. Cash and S. Tessaro, "The locality of searchable symmetric encryption," in *EUROCRYPT 2014*. Springer, 2014, pp. 351–368.

[17] I. Demertzis, D. Papadopoulos, and C. Papamanthou, "Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency," in *Crypto 2018*. Springer, 2018, pp. 371–406.

[18] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption with forward and stronger backward privacy," in *ESORICS 2019*. Springer, 2019, pp. 283–303.

[19] P. Xu, W. Susilo, W. Wang, T. Chen, Q. Wu, K. Liang, and H. Jin, "Rose: Robust searchable encryption with forward and backward security," *TIFS*, vol. 17, pp. 1115–1130, 2022.

[20] D. Yuan, S. Cui, and G. Russello, "We can make mistakes: Fault-tolerant forward private verifiable dynamic searchable symmetric encryption," in *EuroS&P 2022*. IEEE, 2022.

[21] R. Zhang, R. Xue, and L. Liu, "Searchable encryption for healthcare clouds: A survey," *IEEE Transactions on Services Computing*, vol. 11, no. 6, pp. 978–996, 2018.

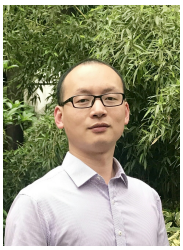
[22] X. Liu, G. Yang, W. Susilo, J. Tonien, R. Chen, and X. Lv, "Message-locked searchable encryption: A new versatile tool for secure cloud storage," *IEEE Transactions on Services Computing*, vol. 15, no. 3, pp. 1664–1677, 2022.
[23] J. Zhang, X. Li, M. Zheng, R. Feng, S. Xu, Z. Hou, and G. Bai, "Verifuzzy: A dynamic verifiable fuzzy search service framework for encrypted cloud data," *IEEE Transactions on Services Computing*, vol. 19, no. 1, pp. 780–793, 2026.
[24] S. Zhu, X. Han, Y. Zhao, and J. Wang, "Balancing service efficiency and data security in hybrid cloud systems: A queueing game approach," *IEEE Transactions on Services Computing*, vol. 19, no. 1, pp. 826–832, 2026.
[25] S. Majumder, S. Ray, M. Dasgupta, P. Bhattacharya, T. R. Gadekallu, and G. Srivastava, "Quanfraud: Quantum state verification scheme for fraud detection in iot-assisted quantum-blockchain networks," *IEEE Transactions on Services Computing*, vol. 19, no. 1, pp. 616–627, 2026.
[26] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *S&P 2000*. IEEE, 2000, pp. 44–55.
[27] E.-J. Goh *et al.*, "Secure indexes." *ePrint*, vol. 2003, p. 216, 2003.
[28] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *CRYPTO 2013*. Springer, 2013, pp. 353–373.
[29] Y. Li, J. Ning, and J. Chen, "Secure and practical wildcard searchable encryption system based on inner product," *IEEE Transactions on Services Computing*, vol. 16, no. 3, pp. 2178–2190, 2023.
[30] C. Guo, X. Chen, Y. Jie, Z. Fu, M. Li, and B. Feng, "Dynamic multi-phrase ranked search over encrypted data with symmetric searchable encryption," *IEEE Transactions on Services Computing*, vol. 13, no. 6, pp. 1034–1044, 2020.
[31] L. Chen, Y. Xue, Y. Mu, L. Zeng, F. Rezaeibagha, and R. H. Deng, "Case-sse: Context-aware semantically extensible searchable symmetric encryption for encrypted cloud data," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1011–1022, 2023.
[32] S.-F. Sun, J. K. Liu, A. Sakzad, R. Steinfeld, and T. H. Yuen, "An efficient non-interactive multi-client searchable encryption with support for boolean queries," in *ESORICS 2016*. Springer, 2016, pp. 154–172.
[33] H. Wang, J. Ning, W. Wu, C. Lin, and K. Zhang, "Ka^{2}2se: Key-aggregation authorized searchable encryption scheme for data sharing in wireless sensor networks," *IEEE Transactions on Services Computing*, vol. 18, no. 1, pp. 226–238, 2025.
[34] Y. Wang, S.-F. Sun, J. Wang, J. K. Liu, and X. Chen, "Achieving searchable encryption scheme with search pattern hidden," *IEEE Transactions on Services Computing*, vol. 15, no. 2, pp. 1012–1025, 2022.
[35] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," in *NDSS 2014*, vol. 71. The Internet Society, 2014.
[36] C. Zuo, S. Sun, J. K. Liu, J. Shao, and J. Pieprzyk, "Dynamic searchable symmetric encryption schemes supporting range queries with forward/backward privacy," *CoRR*, vol. abs/1905.08561, 2019.
[37] S. Patranabis and D. Mukhopadhyay, "Forward and backward private conjunctive searchable symmetric encryption," in *NDSS 2021*. The Internet Society, 2021.
[38] C. Zuo, S.-F. Sun, J. K. Liu, J. Shao, J. Pieprzyk, and L. Xu, "Forward and backward private dsse for range queries," 2020, pp. 328–338.
[39] T. Hoang, A. A. Yavuz, and J. Guajardo, "A secure searchable encryption framework for privacy-critical cloud storage services," *IEEE Transactions on Services Computing*, vol. 14, no. 6, pp. 1675–1689, 2021.
[40] K. Kurosawa and Y. Ohtaki, "Uc-secure searchable symmetric encryption," in *16th International Conference on Financial Cryptography and Data Security, FC 2012*, Kralendijk, Bonaire, Feb. 2012, pp. 285–298.
[41] Q. Chai and G. Gong, "Verifiable symmetric searchable encryption for semi - honest - but - curious cloud servers," in *IEEE International Conference on Communications, ICC 2012*. Ottawa, ON, Canada: IEEE, Jun. 2012, pp. 917–922.
[42] J. Wang, X. Chen, S. Sun, J. K. Liu, M. H. Au, and Z. Zhan, "Towards efficient verifiable conjunctive keyword search for large encrypted database," in *ESORICS 2018*, vol. 11099. Springer, 2018, pp. 83–100.
[43] X. Zhu, J. Zhou, Y. Dai, P. Shen, S. Kasra Kermanshahi, and J. Hu, "A Verifiable and Efficient Symmetric Searchable Encryption Scheme for Dynamic Dataset With Forward and Backward Privacy," *IEEE Trans. Dependable Secure Comput.*, vol. 22, no. 3, pp. 2741–2755, May–Jun 2025.
[44] C. Zhao, R. Du, K. He, J. Chen, J. Li, X. Liu, and J. Ning, "Efficient Verifiable Dynamic Searchable Symmetric Encryption With Forward and Backward Security," *IEEE Internet Things J.*, vol. 12, no. 3, pp. 2633–2645, Feb. 2025.
[45] B. C. Das, N. Datta, A. Majumder, and S. Samajder, "Fault-tolerant verifiable dynamic SSE with forward and backward privacy," *IACR Communications in Cryptology*, vol. 1, no. 4, 2025.
[46] W. Dai, Crypto++ Library 8.2, https://www.cryptopp.com.
[47] Rocksdb 6.6.4, https://github.com/facebook/rocksdb/tree/v6.6.4.
[48] gRPC, https://github.com/grpc/grpc.

Cong Zuo received his Ph.D. degree from Monash University, Australia in 2020. He is currently an associate professor at the school of cyberspace science and technology, Beijing Institute of Technology (BIT), China. Before joining BIT, he was a research fellow at Nanyang Technological University, Singapore from 2021 to 2022. His main research interest is on cybersecurity, in particular, Database Security and Applied Cryptography.

Bingjing Wang is currently a master's degree candidate in the School of Cyberspace Science and Technology, Beijing Institute of Technology (BIT), China. Her research interests focus on cybersecurity and applied cryptography in cloud computing environments, aiming to address privacy preservation and data integrity challenges in distributed systems.

Jianghua Liu received his Ph.D. degree from the School of Information Technology, Deakin University, Australia in 2020. His research interests include cryptography and information security. He is currently an associate Professor with the School of Computer Science and Engineering, Nanjing University of Science and Technology, China. He has published research papers in refereed international journals and conferences, such as IEEE TC, IEEE TSC, IEEE TDSC, ACM TECS, Computers & Security, FGCS, ESORICS 2018 etc.

Shujie Cui is a Lecturer at Monash University in the Faculty of Information Technology. She obtained her PhD degree from the University of Auckland in 2019. Before joining Monash University, She was a Post-Doc researcher in the Large-Scale Data & Systems (LSDS) group in the Department of Computing at Imperial College London, UK. Her main research interests include applied cryptography, information security in cloud computing and distributed systems, trusted execution environments, side-channel attacks, and privacy-preserving machine learning.



Jun Shao received the Ph.D. degree from the Department of Computer Science and Engineering at Shanghai Jiao Tong University, Shanghai, China in 2008. He was a postdoc in the School of Information Sciences and Technology at Pennsylvania State University, USA, from 2008 to 2010. He is currently a professor of the School of Computer and Information Engineering at Zhejiang Gongshang University, Hangzhou, China. His research interests include network security and applied cryptography.

Huaxiong Wang received a Ph.D. in Mathematics from University of Haifa, Israel in 1996 and a Ph.D. in Computer Science from University of Wollongong, Australia in 2001. He has been an associate professor at Nanyang Technological University in Singapore since 2006, where he served as the Head of Division of Mathematical Sciences from 2013 to 2015. Prior to NTU, he held faculty positions at Macquarie University and University of Wollongong in Australia.

Liehuang Zhu received his Ph.D. degree in computer science from Beijing Institute of Technology, Beijing,China, in 2004. He is currently a professor at School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, China. He has published more than100 peer-reviewed journal or conference papers, including a dozen of IEEE/ACM Transactions papers (IEEE TIFS, IEEE TII, IEEE TVT, IEEE TSG, Information Sciences,IEEE Network, Computer and Security, etc.). He has been granted a number of IEEE Best Paper Awards, including IWQoS'17, TrustCom'18. His research interests include blockchain, security protocol analysis and design, cloud computing, etc.

Giovanni Russello is the Head of the School of Computer Science at the University of Auckland, New Zealand. Prof. Giovanni is the Director of the Cyber Security Research Programme, a multi-million project funded by MBIE to improve the cyber security stance of NZ and increase the collaboration between NZ and AU researchers. His research interests include human-centred cyber security, policy-based security systems, privacy and confidentiality in cloud computing, smartphone security, and applied cryptography.