Hermes: Efficient and Secure Multi-Writer Encrypted Database

Tung Le
Virginia Tech
tungle@vt.edu

Thang Hoang
Virginia Tech
thanghoang@vt.edu

***Abstract*—Searchable encryption (SE) enables privacy-preserving keyword search on encrypted data. Public-key SE (PKSE) supports multi-user searches but suffers from high search latency due to expensive public-key operations. Symmetric SE (SSE) offers a sublinear search but is mainly limited to single-user settings. Recently, hybrid SE (HSE) has combined SSE and PKSE to achieve the best of both worlds, including multi-writer encrypted search functionalities, forward privacy, and sublinear search with respect to database size. Despite its advantages, HSE inherits critical security limitations, such as susceptibility to dictionary attacks, and still incurs significant overhead for search access control verification, requiring costly public-key operation invocations (i.e., pairing) across all authorized keywords. Additionally, its search access control component must be rebuilt periodically for forward privacy, imposing substantial writer overhead.**

In this paper, we propose Hermes, a new HSE scheme that addresses the aforementioned security issues in prior HSE designs while maintaining minimal search complexity and user efficiency at the same time. Hermes enables multi-writer encrypted search functionalities and offers forward privacy along with resilience to dictionary attacks. To achieve this, we develop a new identity-based encryption scheme with hidden identity and key-aggregate properties, which could be of independent interest. We also design novel partitioning and epoch encoding techniques in Hermes to minimize search complexity and offer low user overhead in maintaining forward privacy. We conducted intensive experiments to assess and compare the performance of Hermes and its counterpart on commodity hardware. Experimental results showed that Hermes performs search one to two orders of magnitude faster than the state-of-the-art HSE while offering stronger security guarantees to prevent dictionary and injection attacks.

1. Introduction

Data outsourcing services (e.g., Dropbox, Google Drive, OneDrive) have been increasingly prevalent because of their accessibility and convenience. However, user data might be exfiltrated when outsourced to the cloud, which leads to privacy concerns regarding information leakages, especially for sensitive data (e.g., personal and medical records). An adversarial cloud can exploit user data illegitimately, jeopardizing user privacy and the reputation and business of the victim organizations. Although end-to-end encrypted storage systems [9], [10] can mitigate user privacy concerns via data encryption, these systems thwart the capabilities of executing queries (e.g., search) that can be performed on plaintext data.

To enable queries on encrypted data, the concept of Searchable Encryption (SE) [63] has been proposed. The most secure SE paradigm is Dynamic Symmetric SE (DSSE), which constructs an encrypted index that allows a client to perform search (via a keyword trapdoor) or update on their encrypted data without leaking any information about the keywords or data being searched or updated [28], [48], [30], [23]. Many advancements have been made to make DSSE more secure with numerous security properties being achieved such as forward-privacy [17], [56], backward-privacy [37], [56], volume-hiding [46], [69], [14], and search and access pattern obfuscations [42], [41], [61].

Despite its security advantages, DSSE only supports personal search/update functionalities, in which the encrypted data can only be searched/updated by its owner. This strictly limits the practicality and deployability of DSSE in practice, where data is commonly shared and accessed by multiple users. Some attempted to make SSE support multi-user query functionalities; however, most of them require a costly multiparty computation model or a trusted third party to enforce search access controls [29], [51], [71], [52], [50].

To enable multi-user (particularly multi-writer) functionalities without relying on external parties, Wang et al. recently proposed Hybrid SE (HSE) [70], which develops an efficient search access control component atop DSSE using keyword sharing mechanisms in the Public-Key SE (PKSE) model [16]. HSE achieves the best of both SE models, including the sublinear search complexity of DSSE plus the confined search (i.e., the search token size is independent of the number of users) and multi-writer functionalities of PKSE.

Despite its significant potential, integrating PKSE to enable multi-user functionalities in DSSE remains significant security and performance challenges. PKSE is known to be vulnerable to keyword-guessing (dictionary) attacks (KGA) [19] and incurs a high search overhead, requiring to process the entire search access control list [16], [76], [15], [70]. Furthermore, while forward privacy has become a *de facto* standard in DSSE, this feature is not supported by PKSE, leaving the search access control component susceptible to devastating attacks (e.g., file-injection [77]). Previous efforts to (partially) address this vulnerability require the writers to

periodically rebuild their search access control component [70], leading to high user overhead.

Given the critical security and performance challenges in enabling multi-user functionalities in DSSE, we raise the following research question:

Can we make DSSE support multi-writer functionalities with high security (e.g., KGA resiliency, forward privacy) as well as concretely low search complexity and user efficiency?

1.1. Our Contributions

We answer the above question affirmatively with Hermes[1], a new HSE scheme that offers multi-writer encrypted search functionalities and achieves high security and efficiency (asymptotically and concretely) simultaneously.

- **Resiliency to KGA with Minimal Leakage:** Unlike prior HSE [70] and sole PKSE designs, Hermes offers security against KGAs. Hermes can also restrict the search scope to a specified writer subset to prevent unnecessary leakage.
- **Low Reader Bandwidth Overhead:** In Hermes, the reader's search query size is independent of the number of writers, where a single search token encompasses the power to search over *any* number of writers' databases. Therefore, its reader bandwidth overhead is much smaller than other multi-writer encrypted designs (e.g., [29], [50]), whose query size grows linearly in the number of writers. Hermes is desirable for applications involving many writers with disjoint databases (e.g., email [1], messaging [2], scientific collaboration [3], [5]). For these, making query size independent of the number of writers is critical.
- **Sublinear Server Search Complexity:** Hermes achieves sublinear search computation complexity on the server, where it only takes $\mathcal{O}(\sqrt{|W|})$ time to perform an authorized search, where $|W|$ is the number of keywords. This is much more efficient than state-of-the-art multi-writer encrypted search techniques, which require linear processing in the DSSE index of size $\mathcal{O}(W \cdot N)$ [29], [50] (where N is the number of documents) and/or the search access control component [70] of size $\mathcal{O}(|W|)$. We also introduce a novel recursive partitioning technique to further reduce the search complexity to $\mathcal{O}(\log^2 |W|/\log\log|W|)$, at the cost of slightly increasing the search token size from $\mathcal{O}(\lambda^2)$ to $\mathcal{O}(\lambda \log|W|/\log\log|W| + \lambda^2)$.
- **Streamlined Forward Privacy of Search Access Control with Low Writer Overhead.** The search access control component in Hermes achieves forward privacy *directly* during keyword/file updates by the writer, eliminating the need for extra periodic rebuilds. This results in minimal writer overhead, adding only a multiplicative factor of the security parameter λ, compared with the substantial cost of fully rebuilding the entire access control component (typically much larger than λ) in the prior HSE technique [70] to achieve equivalent security. Hermes also eliminates the security risks associated with choosing inappropriate epoch intervals for periodic rebuilds and therefore, achieves stronger security than the previous HSE scheme, which struggles to implement short rebuild epochs. For large databases with many keywords, maintaining short rebuild intervals (e.g., 1 second or 1 minute) may be impractical as the rebuild might not keep up with frequent epoch transitions. This constraint forces the use of longer epochs (e.g., 1 hour or 1 month, as in [70], Sec. 7.1), leaving search and update queries within these extended epochs vulnerable to leakage-abuse and injection attacks.
- **Fully-Fledged Implementation and Evaluation:** We fully implemented Hermes and evaluated its performance on a commodity server. Under real environments, experimental results showed that Hermes performs search up to $164\times$ faster than the state-of-the-art HSE [70]. Our implementation is available at https://github.com/vt-asaplab/Hermes.
- **New IBE Scheme with Key-Aggregate and Hidden Identity:** In this paper, we propose Hidden-Identity Coupling Key-Aggregate Encryption (HICKAE), a new Identity-Based Encryption (IBE) scheme that inherits all desirable properties of an IBE for efficient search access control in multi-writer encrypted search [70] *plus* the additional capability to conceal the identity embedded in the decryption keys. Specifically, HICKAE allows constant-size search query with confined search property, allowing the reader to search on any selected subset of writers' database with a single search token. The search scope restriction to a specific subset reduces unnecessary leakage, which was not supported in most prior multi-writer encrypted search [16], [75], [76], where a token could access *any* writer's database. In scenarios where a corrupt server could act as a writer to inject documents and compromise query confidentiality, confined search complicates such attacks, as readers are more likely to search within their trusted writers' subset rather than across unfamiliar sources. More importantly, HICKAE allows the search token to preserve identity privacy, thereby preventing KGAs.

Table 1 compares Hermes with prior multi-writer encrypted search schemes. To our knowledge, Hermes is the first to simultaneously achieve forward privacy for search access control, low server search complexity, high user efficiency, and resistance to KGAs. Apart from secure communication applications, Hermes can also be useful in EHR management, where multiple physicians (writers) upload patient's EHRs, and the patient (reader) retrieves and searches their medical history. In all these applications, keyword-guessing and injection attacks pose major threats, allowing adversaries to compromise user and data privacy by exploiting leakages in search access control components. Hermes eliminates these risks while also providing sublinear search efficiency and rebuild-free forward privacy, ensuring system scalability, low-latency performance, and strong security for all users involved in real-world deployment.

1.2. Technical Highlights

Brief Overview of HSE [70]. HSE consists of two main components: (*i*) a DSSE-based encrypted index containing encrypted keyword-file pairs, and (*ii*) a PKSE-based search access control component containing encrypted sharing

1. Hermes stands for Highly-efficient and secure multi-writer encrypted search.

Table 1: Comparison of Hermes with prior multi-writer encrypted search systems.

Scheme	Search Complexity			Update Complexity¶			Forward Privacy	No Rebuild	KGA Resiliency
	Reader	Server	Reader Comm. (In + Out)	Writer	Server	Comm.			
PEKS [16]	$\mathcal{O}(1)$	$\mathcal{O}(\|\mathsf{DB}\|^{\dagger})$	$\mathcal{O}(r_w+\lambda)$	$\mathcal{O}(\|W_u\|)$	$\mathcal{O}(1)$	$\mathcal{O}_\lambda(\|W_u\|)$	✗	_	✗
NTRU-PEKS [15]	$\mathcal{O}(1)$	$\mathcal{O}(\|\mathsf{DB}\|)$	$\mathcal{O}(r_w+\lambda)$	$\mathcal{O}(\|W_u\|)$	$\mathcal{O}(1)$	$\mathcal{O}_\lambda(\|W_u\|)$	✗	_	✗
PAUKS [52]	$\mathcal{O}(1)$	$\mathcal{O}(\|W\|^{\dagger}+{d_w}^{\dagger})$	$\mathcal{O}(r_w+\lambda)$	$\mathcal{O}(\|W_u\|)$	$\mathcal{O}(1)$	$\mathcal{O}_\lambda(\|W_u\|)$	✗	_	✓
AESM^2 [71]	$\mathcal{O}(1)$	$\mathcal{O}(\|W\|^{\dagger}+d_w)$	$\mathcal{O}(r_w+\lambda)$	$\mathcal{O}(\|W_u\|)$	$\mathcal{O}(1)$	$\mathcal{O}_\lambda(\|W_u\|)$	✗	_	✓
FP-HSE [70]	$\mathcal{O}(1)$	$\mathcal{O}(\|W\|^{\dagger}+d_w)$	$\mathcal{O}(r_w+\lambda)$	$\mathcal{O}(\|W\|^{\dagger})$	$\mathcal{O}(1)$	$\mathcal{O}_\lambda(\|W\|)$	✓	✗	✗
Our Hermes	$\mathcal{O}(\lambda)$	$\mathcal{O}(\lambda\sqrt{\|W\|^{\dagger}+d_w})$	$\mathcal{O}(r_w+\lambda^2)$	$\mathcal{O}(\|W_u\|^{\dagger})$	$\mathcal{O}(1)$	$\mathcal{O}_{\lambda^2}(\|W_u\|)$	✓	✓	✓
Our Hermes$^+$	$\mathcal{O}(\frac{\log\|W\|}{\log\log\|W\|}+\lambda)$	$\mathcal{O}(\frac{\log^2\|W\|}{\log\log\|W\|}{}^{\dagger}+d_w)$	$\mathcal{O}(r_w+(\frac{\lambda\log\|W\|}{\log\log\|W\|}+\lambda^2))$	$\mathcal{O}_\lambda(\|W_u\|^{\dagger})$	$\mathcal{O}(1)$	$\mathcal{O}_{\lambda^2}(\|W_u\|)$	✓	✓	✓

DB: The whole keyword-file pairs database; W: The set of unique keywords over *all* documents (keyword universe); W_u: The set of updated (added) keywords.
λ: Security parameter; r_w: Number of documents matched keyword search; d_w: Number of updates of keyword w; †: Public-key pairing operation.
¶: FP-HSE requires a rebuild operation in update to maintain forward privacy. See §6 for detailed experiments.
For simplicity, we use $\mathcal{O}_\lambda(\cdot)$ (similarly for $\mathcal{O}_{\lambda^2}(\cdot)$) to hide a multiplicative factor of the security parameter in the update complexity.

tokens. The keyword-file pairs are encrypted by symmetric encryption and can be decrypted using a valid keyword trapdoor, as in standard DSSE. Meanwhile, each sharing token is created using an IBE called Identity-Coupling Key-Aggregate Encryption (ICKAE), where the DSSE keyword trapdoor is encrypted using the reader's public key, and the authorized keyword serves as the identity. To achieve the key-aggregate property for confined search, ICKAE relies on a Structured Reference String (SRS), where the writer uses its deterministically assigned element in the SRS to create sharing tokens. When the reader searches selected writers' databases, they generate an aggregated decryption key using the searched keyword as the identity and the assigned SRS element of the selected writers. The server then uses this aggregated key to attempt decryption of *all* sharing tokens from the selected writers, retrieving the keyword trapdoor, which is subsequently used to decrypt the DSSE-encrypted index and obtain the search results.

There are three main limitations in the current HSE design as follows.

- *Keyword-Guessing Vulnerability*: HSE uses ICKAE to enforce search access control. Although ICKAE protects identity (i.e., keyword) confidentiality in the ciphertext (i.e., anonymity), it unfortunately does not protect the identity confidentiality associated with the aggregated decryption key due to its public-key encryption structure. Since the identity space is finite and can be known by the adversary (e.g., dictionary), the adversary can create ciphertext under its chosen identity using the public key of the reader, then try matching it with the given aggregated key (see §2.2 for more details on how this attack works).
- *Costly Periodic Rebuild for Forward Privacy*: HSE enables forward privacy for the search access control component by concatenating the keyword with a single epoch value as the identity in IBE when creating sharing tokens. As a result, whenever a new epoch starts, the writers need to update/rebuild their existing sharing tokens to make them consistent with the current epoch so that the reader query can search on their database. This rebuild complexity is linear in the number of sharing tokens, which incurs $\mathcal{O}(|W|)$ number of public-key operations and $\mathcal{O}(|W|)$ communication cost, where $|W|$ is the number of authorized keywords.
- *Linear Server Search Complexity*: In HSE, the server needs to test the reader's search token (i.e., the aggregated decryption key) against all the writer's sharing tokens to obtain the DSSE trapdoor, leading to $\mathcal{O}(|W|)$ computational complexity. This cost is extremely high because each test incurs two expensive public-key pairing operations.

Can we address all these challenges?

Idea 1: KGA-Resilience via Hidden-Identity IBE. KGA stems from the leakage of the identity in the aggregated decryption key of the underlying IBE used for search access control. Therefore, to address this, we design HICKAE, which offers the same properties as ICKAE (e.g., key-aggregate, anonymity) but can prevent the chosen-identity attack that could exploit the decryption key. The core idea is to have the encryptor embed a secret into the ciphertexts and only the decryptor who knows the private key and the encryptor secret can generate the aggregated key. The challenge is to ensure only a single key is needed to decrypt all ciphertexts (associated with the same identity but different secrets) created by different encryptors. We tackle this by securely establishing correlation values between any two encryptors (see §4.2 for details). When HICKAE is used for search access control, its ability to conceal the identity in the decryption key enhances Hermes's resilience to KGAs.

Idea 2: Forward Privacy without Rebuild via Epoch Encoding. To avoid costly rebuilds for forward privacy, we propose a new technique to encode the epoch information in a way that it can restrict the capability of the reader's search query, allowing queries issued at the current epoch to only search over encrypted sharing tokens created *at or before* that epoch. When the writers perform updates at a new epoch e', they generate new encrypted sharing tokens using the encoded string of e'. Existing search queries issued at epochs $< e'$ become invalid for accessing sharing tokens created at epochs $\geq e'$, thus achieving forward privacy. Since search queries in Hermes are built from aggregated keys in HICKAE, we encode epochs as part of the identity, such that aggregated keys used for decrypting current ciphertexts do not compromise the privacy of future ones, while remaining valid for all created at or before the current epoch. This strategy offers a reasonable trade-off, offering fast search and forward privacy for the search access control component without requiring rebuilds, and only slightly increasing server communication and storage by a factor of $\mathcal{O}(\lambda)$.

Idea 3: Sublinear Search Complexity via Recursive Partitioning. To reduce server complexity, we securely cluster the search access control component into partitions.

During a search, the partition address corresponding to the queried keyword is revealed, allowing to test only the encrypted sharing tokens within that partition with the search queries to identify a match. In particular, the authorized keyword sharing tokens are distributed uniformly to $\sqrt{|W|}$ partitions each containing $\mathcal{O}(\sqrt{|W|})$ tokens. To search for a keyword, the server only needs to perform cryptographic tests within the specified partition, rather than over the entire search access control component. Each partition address is computed using a keyed pseudorandom function (PRF) only known by the writer. Thus, even if the adversary learns the partition address, they gain no information about the queried keyword, and therefore, the query confidentiality is preserved. To let the server know which partition the searched keyword belongs to, the reader sends a complementary search token that allows the server to identify the correct partition (via cryptographic testing) before matching the keyword query with the sharing tokens. Although this strategy incurs an additional computation cost $\mathcal{O}(\sqrt{|W|})$ for partition identification, it still maintains a constant search query of size $\mathcal{O}(\lambda^2)$. It is worth noting that simple partitioning can lead to KGA vulnerabilities due to the correlation between partition indices and keywords. To address this, we integrate this partitioning strategy with our proposed HICKAE scheme to achieve sublinear search complexity while simultaneously preserving KGA resilience. To our knowledge, no prior work has achieved sublinear search with KGA resilience through partitioning. Finally, we show that this partitioning strategy can be applied recursively to further reduce the computational search complexity to $\mathcal{O}(\log^2 |W|/\log\log|W|)$ with search query of size $\mathcal{O}(\lambda \log|W|/\log\log|W| + \lambda^2)$ as a trade-off.

2. Preliminaries

Notation. We denote by λ the security parameter and by $\mathbb{Z}_q$ a group of integers modulo q. We denote $[n]$ as $\{1,\ldots,n\}$ and $[x,y]$ as $\{x, x+1, \ldots, y\}$. $x \xleftarrow{\$} [n]$ means x is selected uniformly at random from $[n]$. Bold small letters (e.g., $\mathbf{t}$) denote vectors, where $|\mathbf{t}|$ denotes the dimension of $\mathbf{t}$. Let $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t)$ be a tuple of cyclic groups of prime order q and equipped with pairing e: $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$. Let $p = q-1$. We denote $[1]_1 \in \mathbb{G}_1$ and $[1]_2 \in \mathbb{G}_2$ as generators in their groups, respectively. We define $[1]_t \leftarrow e([1]_1, [1]_2)$ as a generator of $\mathbb{G}_t$. For $i \in \{1,2,t\}$ and $x \in \mathbb{Z}_q$, $[x]_i \in \mathbb{G}_i$ denotes the group element whose discrete logarithm base $[1]_i$ is x. The operation between two group elements $[x]_i \in \mathbb{G}_i$ and $[y]_i \in \mathbb{G}_i$ is written as addition, *i.e.*, $[x]_i + [y]_i \leftarrow [x+y]_i \in \mathbb{G}_i$. The pairing is expressed as multiplication, *i.e.*, $[x]_1[y]_2 \leftarrow e([x]_1, [y]_2) = [xy]_t \in \mathbb{G}_t$.

2.1. Searchable Encryption

Searchable Encryption permits a user to perform privacy-preserving keyword searches over encrypted documents without leaking the plaintext of the keyword and documents to the storage server. We recall DSSE, which enables not only keyword search but also document update.

n	Number of classes/writers
$[n]$	$\{1,\ldots,n\}$
$[x]_i$	Group element in $\mathbb{G}_i$: $[x]_i = x[1]_i$ with $i \in \{1,2,t\}$
$G(.), H(.)$	Cryptographic hash functions
$\mathsf{PTkn}/\mathsf{WTkn}$	Encrypted partition/keyword tokens set
EIDX	Encrypted search index (DSSE)
ESTkn	Search access control $\mathsf{ESTkn} = (\mathsf{PTkn}, \mathsf{WTkn})$
$\hat{\sigma}/\sigma'_i$	The reader trapdoor/the writer i's secret, $i \in [n]$
σ_i	The writer i's shared secret $\sigma_i = \hat{\sigma}^i + \sigma'_i$, $i \in [n]$
Σ	Set of shared secrets $\Sigma = \{\sigma_i\}_{i\in[n]}$
Corr	Correlation set $\mathsf{Corr} = \{[\Delta_{i,j}]_1\}_{i,j\in[n]\wedge i\neq j}$
$(\mathsf{rpk}, \mathsf{rsk})$	The reader's public and private key pair
κ_i	The writer i's DSSE secret key
$[\mathsf{ek}_i]_2$	The writer i's class-binding key
wsk_i	The writer i's secret key $\mathsf{wsk}_i = (\kappa_i, [\mathsf{ek}_i]_2)$
S	A writer subset $S \subseteq [n]$
e	Epoch number
$e(\mathbf{t})$	The bijection mapping tag $\mathbf{t}$ to epoch e
$\mathbf{t}(e)$	The inverse function mapping epoch e to tag $\mathbf{t}$
$\Gamma_{\mathbf{t}}$	The smallest set of tags including a prefix of all $\mathbf{t}' \succeq \mathbf{t}$
P_e	Set of prefixes of encoded epoch $\mathbf{t}(e)$
W	Active keyword set
$\mathfrak{s}/\mathfrak{u}$	Search token/update token

Table 2: Summary of notation

Definition 1 (DSSE). *A* DSSE *scheme is a tuple of PPT algorithms defined as follows:*

- $(\kappa, \mathsf{EIDX}) \leftarrow \mathsf{Setup}(1^\lambda)$*: Given a security parameter* λ*, it outputs a secret key* κ*, and an encrypted index* EIDX*.*
- $\mathfrak{s} \leftarrow \mathsf{SrchTkn}(\kappa, w)$*: Given a secret key* κ*, and a keyword* w*, it outputs a keyword trapdoor* $\mathfrak{s}$*.*
- $\mathcal{R} \leftarrow \mathsf{Srch}(\mathfrak{s}, \mathsf{EIDX})$*: Given a trapdoor* $\mathfrak{s}$ *and a DSSE encrypted index* EIDX*, it outputs a result* $\mathcal{R}$*.*
- $\mathfrak{u} \leftarrow \mathsf{UpdtTkn}(\kappa, \mathsf{op}, w, f)$*: Given a secret key* κ*, an operation* $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$ *and a keyword-file pair* (w, f) *to be added/deleted, it outputs an update token* $\mathfrak{u}$*.*
- $\mathsf{EIDX}' \leftarrow \mathsf{Updt}(\mathfrak{u}, \mathsf{EIDX})$*: Given an update token* $\mathfrak{u}$ *and an encrypted index* EIDX*, it outputs an updated index* EIDX'*.*

2.2. ID-Coupling KAE

KAE [26] is a Public-Key Encryption (PKE) that permits the decryption of multiple ciphertexts created by different encryptors (called *classes*) using an aggregated key. To enable efficient multi-writer encrypted search, Wang et al. [70] proposed ID-Coupling KAE (ICKAE) that upgrades KAE from PKE to IBE by embedding a unique identity (e.g., a keyword) into the ciphertext and decryption key, such that the decryption key for a class subset and a specific identity can only decrypt the ciphertexts created by the classes in the subset and the same identity.

Let $\mathsf{sk} = (\gamma, \delta)$ be the private key with the corresponding public key $\mathsf{pk} = ([\delta]_2, [\gamma]_2)$. Let $([\alpha^i]_1, [\alpha^i]_2, [\alpha^{n+1}]_t)$, with $i \in [2n] \setminus \{n+1\}$, be the SRS public parameters. To encrypt a message m with the corresponding identity id, class i computes $c \leftarrow ([c_1]_2, [c_2]_2, c_3)$, where $[c_1]_2 \leftarrow [r]_2$, $[c_2]_2 \leftarrow r([\gamma]_2 + [\alpha^i]_2)$, and $c_3 \leftarrow m \oplus G(r([\alpha^{n+1}]_t - r[h]_1[\delta]_2))$, with $[h]_1 \leftarrow H(\mathsf{id})$. Here, $H : \{0,1\}^* \to \mathbb{G}_1$ and $G : \mathbb{G}_t \to \{0,1\}^\lambda$ are two cryptographic hash functions, and $r \in \mathbb{Z}_q$ is random. A single key $[k]_1 = \gamma \sum_{j\in S}[\alpha^{n+1-j}]_1 + \delta[h]_1$ can allow decrypting multiple ciphertexts with the

corresponding id, created by multiple classes $S \subseteq [n]$, as $m = c_3 \oplus G([u]_t)$, where $[u]_t = \sum_{j \in S} [\alpha^{n+1-j}]_1 [c_2]_2 - \left([k]_1 + \sum_{j \in S \setminus \{i\}} [\alpha^{n+1+i-j}]_1\right) [c_1]_2 = r[\alpha^{n+1}]_t - r[h]_1[\delta]_2$

KGA Vulnerability. In ICKAE, the decryption key leaks the embedded identity id, leading to KGA when used for multi-writer encrypted search. Specifically, the keyword w and its DSSE trapdoor τ_w are treated as the identity (i.e., $w = \mathsf{id}$) and the message (i.e., $\tau_w = m$) in ICKAE, respectively. With the key $[k]_1 = \gamma \sum_{j \in S} [\alpha^{n+1-j}]_1 + \delta[h]_1$ and the public parameters $\{[\alpha^{n+1-j}]_1\}_{j \in S}$, the adversary can compute $[k']_t = [k]_1[1]_2 - [\gamma]_2 \sum_{j \in S} [\alpha^{n+1-j}]_1 = [h]_1[\delta]_2$ and guess which keyword is being associated with the decryption key by checking if $[k']_t \stackrel{?}{=} [h']_1[\delta]_2$, where $[h']_1 \leftarrow H(w')$ for each chosen keyword w'.

Table 2 summarizes the symbols and notation in our schemes.

3. Models

3.1. System Model

Our system includes three entities: a reader, n writers, and a server. Each writer $i \in [n]$ owns an independent document collection and shares it with the reader. Each writer cannot update, delete, or modify the data of other writers. The reader can perform privacy-preserving keyword search over document collections of a writer subset $S \subseteq [n]$. We consider the reader and writers as independent parties and they do not have to communicate with each other during search and update operations. Our system is a Multi-writer Searchable Encryption (MSE) scheme with KGA-resiliency as follows.

Definition 2 (MSE). *An MSE scheme with KGA-resiliency is a tuple of PPT algorithms defined as follows:*

- $(\mathsf{rpk}, \mathsf{rsk}, \hat{\sigma}, \mathsf{pp}) \leftarrow \mathsf{RSetup}(1^\lambda, n)$*: Executed by the reader given a security parameter λ, a number of writers n, it outputs a public and private key pair* $(\mathsf{rpk}, \mathsf{rsk})$*, a reader trapdoor $\hat{\sigma}$, and public parameters* pp*.*
- $(\mathsf{wsk}_i, \sigma'_i, \mathsf{EIDX}_i, \mathsf{ESTkn}_i) \leftarrow \mathsf{WSetup}(1^\lambda, i, \mathsf{pp})$*: Executed by a writer given its identifier i, a security parameter λ, and the public parameters* pp*, it outputs a writer secret key* wsk_i*, a secret σ'_i, an encrypted index* EIDX_i*, and a search access control* ESTkn_i*.*
- $(\Sigma, \mathsf{Corr}) \leftarrow \mathsf{RPrep}(\mathsf{rsk}, \hat{\sigma}, \sigma')$*: Executed by the reader given its private key* rsk *and trapdoor $\hat{\sigma}$, and the writer secrets $\sigma' = \{\sigma'_i\}_{i \in [n]}$, it outputs a set of shared secrets Σ and a set of writer correlations* Corr*.*
- $\mathfrak{s} \leftarrow \mathsf{SrchTkn}(\mathsf{rsk}, S, \Sigma, w)$*: Executed by the reader given a private key* rsk*, a writer subset $S \subseteq [n]$, a shared secret set Σ, and a keyword w, it outputs a search token $\mathfrak{s}$.*
- $\mathcal{R} \leftarrow \mathsf{Srch}(\mathfrak{s}, S, \mathsf{EIDX}, \mathsf{ESTkn}, \mathsf{Corr})$*: Executed by the server given a search token $\mathfrak{s}$, a writer subset S, the writers' encrypted index $\mathsf{EIDX} = \{\mathsf{EIDX}_i\}_{i \in [n]}$ and their search access control $\mathsf{ESTkn} = \{\mathsf{ESTkn}_i\}_{i \in [n]}$, and the writer correlation set* Corr*, it outputs the search result $\mathcal{R}$.*
- $\mathfrak{u}_i \leftarrow \mathsf{UpdtTkn}(\mathsf{rpk}, \mathsf{wsk}_i, i, \mathsf{op}, w, f)$*: Executed by the writer given its identifier i and secret key* wsk_i*, the reader's public key* rpk*, an operation* $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$*, and a keyword-file pair (w, f), it outputs an update token $\mathfrak{u}_i$.*
- $(\mathsf{EIDX}'_i, \mathsf{ESTkn}'_i) \leftarrow \mathsf{Updt}(\mathfrak{u}_i, \mathsf{EIDX}_i, \mathsf{ESTkn}_i)$*: Executed by the server given an update token $\mathfrak{u}_i$, an encrypted index EIDX_i and a search access control ESTkn_i of writer i, it outputs the updated index EIDX'_i and the updated search access control ESTkn'_i.*

$\mathsf{IND}^b_{\mathsf{MSE},\mathcal{A},\mathcal{L}}(1^\lambda)$:
1: $n \leftarrow \mathcal{A}(1^\lambda)$; $(\mathsf{rpk}, \mathsf{rsk}, \hat{\sigma}, \mathsf{pp}) \leftarrow \mathsf{RSetup}(1^\lambda)$
2: **foreach** $i \in [n]$
3: $\quad (\mathsf{wsk}_i, \sigma'_i, \mathsf{EIDX}_i, \mathsf{ESTkn}_i) \leftarrow \mathsf{WSetup}(1^\lambda, i, \mathsf{pp})$
4: $(\Sigma, \mathsf{Corr}) \leftarrow \mathsf{RPrep}(\mathsf{rsk}, \hat{\sigma}, \sigma')$, where $\sigma' \leftarrow \{\sigma'_i\}_{i \in [n]}$
5: $\mathsf{EIDX} \leftarrow \{\mathsf{EIDX}_i\}_{i \in [n]}$; $\mathsf{ESTkn} \leftarrow \{\mathsf{ESTkn}_i\}_{i \in [n]}$
6: $\mathcal{H}_0 \leftarrow \{\varnothing\}$; $\mathcal{H}_1 \leftarrow \{\varnothing\}$; $\mathcal{O} \leftarrow \{\mathsf{Crpt}\mathcal{O}_b, \mathsf{Srch}\mathcal{O}_b, \mathsf{Updt}\mathcal{O}_b\}$
7: $b' \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{st}_{\mathcal{A}}, \mathsf{pp}, \mathsf{rpk}, \mathsf{EIDX}, \mathsf{ESTkn})$
8: **return** b'

$\mathsf{Srch}\mathcal{O}_b(\{w_k, S_k\}_{k \in \{0,1\}})$:
9: **if** $\mathcal{L}^{\mathsf{Srch}}_{\mathcal{H}_0}(w_0, S_0) = \mathcal{L}^{\mathsf{Srch}}_{\mathcal{H}_1}(w_1, S_1)$ **then**
10: $\quad \forall\, k \in \{0,1\}, \mathcal{H}_k \leftarrow \mathcal{H}_k \cup (\mathsf{Srch}, w_k, S_k)$
11: $\quad$ **return** $\mathsf{SrchTkn}(\mathsf{rsk}, S_b, \Sigma, w_b)$
12: **else return** $\perp$

$\mathsf{Updt}\mathcal{O}_b(\{i_k, \mathsf{op}_k, w_k, f_k\}_{k \in \{0,1\}})$:
13: **if** $\mathcal{L}^{\mathsf{Updt}}_{\mathcal{H}_0}(i_0, \mathsf{op}_0, w_0, f_0) = \mathcal{L}^{\mathsf{Updt}}_{\mathcal{H}_1}(i_1, \mathsf{op}_1, w_1, f_1)$ **then**
14: $\quad \forall\, k \in \{0,1\}, \mathcal{H}_k \leftarrow \mathcal{H}_k \cup (\mathsf{Updt}, i_k, \mathsf{op}_k, w_k, f_k)$
15: $\quad$ **return** $\mathsf{UpdtTkn}(\mathsf{rpk}, \mathsf{wsk}_{i_b}, i_b, \mathsf{op}_b, w_b, f_b)$
16: **else return** $\perp$

$\mathsf{Crpt}\mathcal{O}_b(i_0, i_1)$:
17: **if** $\mathcal{L}^{\mathsf{Crpt}}_{\mathcal{H}_0}(i_0) = \mathcal{L}^{\mathsf{Crpt}}_{\mathcal{H}_1}(i_1)$ **then**
18: $\quad \forall\, k \in \{0,1\}, \mathcal{H}_k \leftarrow \mathcal{H}_k \cup (\mathsf{Crpt}, i_k)$
19: $\quad$ **return** wsk_{i_b}
20: **else return** $\perp$

Figure 1: Security Game for MSE.

Definition 3 (Correctness of MSE). *For all parameters λ and n, all* $(\mathsf{rpk}, \mathsf{rsk}, \hat{\sigma}, \mathsf{pp}) \leftarrow \mathsf{RSetup}(1^\lambda, n)$*, all* $(\mathsf{wsk}_i, \sigma'_i, \mathsf{EIDX}_i, \mathsf{ESTkn}_i) \leftarrow \mathsf{WSetup}(1^\lambda, i, \mathsf{pp})$ *with $i \in [n]$, all* $(\Sigma, \mathsf{Corr}) \leftarrow \mathsf{PRep}(\mathsf{rsk}, \hat{\sigma}, \sigma')$*, where $\sigma' = \{\sigma'_i\}_{i \in [n]}$, and all sequences of* Srch*,* Updt *operations over $\{(\mathsf{EIDX}_i, \mathsf{ESTkn}_i)\}_{i \in [n]}$ using tokens generated respectively from* $\mathsf{SrchTkn}(\mathsf{rsk}, S, \Sigma, w)$*, and* $\mathsf{UpdtTkn}(\mathsf{rpk}, \mathsf{wsk}_i, i, \mathsf{op}, w, f)$*,* Srch *returns the correct results w.r.t. the inputs (i, op, w, f) of* $\mathsf{UpdtTkn}$ *when $i \in S$, except with negligible probability in λ.*

3.2. Threat and Security Models

We consider the standard threat model of MSE, where the server and some of the writer(s) can be corrupt while the reader is honest [70]. We assume the adversary is semi-honest, in which it is curious about the queries of the reader (i.e., keyword search) and writers (e.g., document update) but follows the protocols faithfully. For non-trivial KGA, we consider that the server can collude with some of the

writer(s) as long as the reader does not search on the index of the corrupt writers.

The adversary can issue a sequence of oracle queries of three kinds: (*i*) corruption query, which returns the secret key κ_i and the identity-binding key ek_i of a specific writer i; (*ii*) search query, which returns the search token $\mathfrak{s}$ of a specified keyword w under a chosen writer subset S; and (*iii*) update query, which returns the update token $\mathfrak{u}$ of a specified update tuple from a certain writer. The adversary could issue queries depending on prior outcomes.

To define the semantic security of MSE, we present the history notion with non-singularity that is extended from single-user search [28] to capture historical leakage from corrupt writers in the context of multi-user search [70].

Definition 4 (Non-Singular History). *A history of* MSE *is a query sequence* $\mathcal{H} = \{\mathsf{Hist}_t\}$*, where sequence number* t *denotes the timestamp when the query happens and* $\mathsf{Hist}_t \in \{(\mathsf{Crpt}, i), (\mathsf{Srch}, w, S), (\mathsf{Updt}, i, \mathsf{op}, w, f)\}$.

A history $\mathcal{H}$ *is* non-singular *if there exists at least one history* $\mathcal{H}' \neq \mathcal{H}$ *can be found in polynomial-time given* $\mathcal{L}_{\mathcal{H}}$ *such that* $\mathcal{L}_{\mathcal{H}} = \mathcal{L}_{\mathcal{H}'}$.

We introduce a leakage function family $\mathcal{L}_{\mathcal{H}} = \{\mathcal{L}^{\mathsf{Setup}}_{\mathcal{H}}, \mathcal{L}^{\mathsf{Srch}}_{\mathcal{H}}, \mathcal{L}^{\mathsf{Updt}}_{\mathcal{H}}, \mathcal{L}^{\mathsf{Crpt}}_{\mathcal{H}}\}$ to control exactly the information of history $\mathcal{H}$ leaked during setup, search, update, and corruption, respectively. When an oracle is queried for the t-th operation, any function in $\mathcal{L}_{\mathcal{H}}$ is instantiated with $\mathcal{H}$ being the history consisting of the first $(t-1)$ operations and with the t-th operation as a function input. It records the leakage incurred due to the last operation while taking all historical operations into consideration. We omit $\mathcal{H}$ if there is no ambiguity. Before any query (i.e., $\mathcal{H} = \{\varnothing\}$), $\mathcal{L} = \mathcal{L}^{\mathsf{Setup}}$.

Corruption Leakage. We define $\mathcal{I}_c = \{i : (\mathsf{Crpt}, i) \in \mathcal{H}\}$ as the set of corrupt writers. To capture the corruption leakage, for any writer $i \in [n]$, we introduce a function $\mathsf{UpdtBy}(i)$ based on history $\mathcal{H}$, which lists all updates by i in the history: $\mathsf{UpdtBy}(i) = \{\mathsf{Hist}_t : \mathsf{Hist}_t = (\mathsf{Updt}, i, \mathsf{op}, w, f) \in \mathcal{H}\}$.

We state the formal security definition of MSE as follows.

Definition 5 (Adaptive Security of MSE). *For all* PPT *adversary* $\mathcal{A}$ *and the game* $\mathsf{IND}^b_{\mathsf{MSE},\mathcal{A},\mathcal{L}}(1^\lambda)$ *defined in Figure 1,* MSE *is* $\mathcal{L}$*-adaptively-secure if:*
$|Pr[\mathsf{IND}^0_{\mathsf{MSE},\mathcal{A},\mathcal{L}}(1^\lambda) = 1] - Pr[\mathsf{IND}^1_{\mathsf{MSE},\mathcal{A},\mathcal{L}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda)$

Apart from semantic security, another vital security property in SE is *forward privacy*, which prevents the server from inferring updated documents/keywords or executing injection attacks [77]. While forward privacy in (single-user) DSSE has been clearly defined and well-studied, where a single entity (data owner) performs both search and update [18], achieving forward privacy in multi-user SE is more challenging. This is because searches and updates are performed by separate entities (reader and writer, *resp.*) that do not communicate during the operations. In this context, forward privacy can only be defined in terms of epochs, where the update leakage is restricted to a fixed time interval. If no search has been issued for the keyword being updated within the same epoch, the update leakage remains the same as the standard forward privacy.

$\underline{\mathsf{Setup}(1^\lambda)}$:
$n \leftarrow \mathcal{A}(1^\lambda)$; $(\hat{\sigma}, \mathsf{pp}) \leftarrow \mathsf{HICKAE.Setup}(1^\lambda, n)$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{HICKAE.KeyGen}(1^\lambda)$
$\forall i \in [n]$, $(\sigma'_i, \mathsf{ek}_i) \leftarrow \mathsf{HICKAE.IGen}(1^\lambda, i, \mathsf{pp})$
$\sigma' \leftarrow \{\sigma'_i\}_{i\in[n]}$; $(\Sigma, \mathsf{Corr}) \leftarrow \mathsf{HICKAE.Prep}(\mathsf{sk}, \hat{\sigma}, \sigma')$
$\mathsf{akSet} \leftarrow \{\varnothing\}$; $\mathsf{ctSet} \leftarrow \{\varnothing\}$
return $(n, \mathsf{pk}, \mathsf{sk}, \Sigma, \mathsf{Corr}, \mathsf{akSet}, \mathsf{ctSet})$

$\underline{\mathsf{Ext}\mathcal{O}(S, \mathsf{id})}$:
foreach $i \in S$
  $\mathsf{akSet} \leftarrow \mathsf{akSet} \cup \{(i, \mathsf{id})\}$
return $\mathsf{ak} \leftarrow \mathsf{HICKAE.Ext}(\mathsf{sk}, S, \Sigma, \mathsf{id})$

$\underline{\mathsf{Enc}\mathcal{O}(i, \mathsf{id}, m)}$:
$\mathsf{ctSet} \leftarrow \mathsf{ctSet} \cup \{(i, \mathsf{id})\}$
return $c \leftarrow \mathsf{HICKAE.Enc}(\mathsf{pk}, \mathsf{ek}_i, \mathsf{id}, m)$

$\underline{\mathsf{IND\text{-}CPA}^b_{\mathsf{HICKAE},\mathcal{A}}(1^\lambda)}$:
$(n, \mathsf{pk}, \mathsf{sk}, \Sigma, \mathsf{Corr}, \mathsf{akSet}, \mathsf{ctSet}) \leftarrow \mathsf{Setup}(1^\lambda)$
$i^* = \perp$; $\mathsf{id}^* = \perp$
$(\mathsf{st}, m_0, m_1, i^*, \mathsf{id}^*) \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{pk})$
if $(i^*, \mathsf{id}^*) \in \mathsf{akSet}$ **then return** $\perp$
$c^* \leftarrow \mathsf{HICKAE.Enc}(\mathsf{pk}, \mathsf{ek}_{i^*}, \mathsf{id}^*, m_b)$
return $b' \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{st}, c^*)$

$\underline{\mathsf{IND\text{-}ANON}^b_{\mathsf{HICKAE},\mathcal{A}}(1^\lambda)}$:
$(n, \mathsf{pk}, \mathsf{sk}, \Sigma, \mathsf{Corr}, \mathsf{akSet}, \mathsf{ctSet}) \leftarrow \mathsf{Setup}(1^\lambda)$
$i^* = \perp$; $\mathsf{id}^*_0 = \mathsf{id}^*_1 = \perp$
$(\mathsf{st}, m, i^*, \mathsf{id}^*_0, \mathsf{id}^*_1) \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{pk})$
if $(i^*, \mathsf{id}^*_0) \in \mathsf{akSet} \ \vee (i^*, \mathsf{id}^*_1) \in \mathsf{akSet}$ **then return** $\perp$
$c^* \leftarrow \mathsf{HICKAE.Enc}(\mathsf{pk}, \mathsf{ek}_{i^*}, \mathsf{id}^*_b, m)$
return $b' \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{st}, c^*)$

$\underline{\mathsf{IND\text{-}CIA}^b_{\mathsf{HICKAE},\mathcal{A}}(1^\lambda)}$:
$(n, \mathsf{pk}, \mathsf{sk}, \Sigma, \mathsf{Corr}, \mathsf{akSet}, \mathsf{ctSet}) \leftarrow \mathsf{Setup}(1^\lambda)$
$i^* = \perp$; $\mathsf{id}^*_0 = \mathsf{id}^*_1 = \perp$
$(\mathsf{st}, i^*, \mathsf{id}^*_0, \mathsf{id}^*_1) \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{pk})$
if $(i^*, \mathsf{id}^*_0) \in \mathsf{ctSet} \ \vee (i^*, \mathsf{id}^*_1) \in \mathsf{ctSet}$ **then return** $\perp$
$\mathsf{ak}^* \leftarrow \mathsf{HICKAE.Ext}(\mathsf{sk}, \{i^*\}, \Sigma, \mathsf{id}^*_b)$
return $b' \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{st}, \mathsf{ak}^*)$

Figure 2: Security Game for HICKAE.

Definition 6 (Epoch-Based Forward Privacy of MSE). *Let* $W_{\mathsf{Srch}}(i, e)$ *be the set of keywords that has been searched over any writer subset containing* i *during epoch* e*, i.e.,* $W_{\mathsf{Srch}}(i, e) = \{w : (\mathsf{Srch}, w, S, e) \in \mathcal{H} \wedge i \in S\}$.

An $\mathcal{L}$*-adaptively-secure* MSE *is epoch-based forward-private if the update leakage* $\mathcal{L}^{\mathsf{Updt}}(i, \mathsf{op}, w, f)$ *of any update* (op, w, f) *by any writer* $i \notin \mathcal{I}_c$ *can be written as* $\mathcal{L}'(i, \mathsf{op}, f)$ *provided that* $w \notin W_{\mathsf{Srch}}(i, e)$*, where* $\mathcal{L}'$ *is stateless.*

It is important to note that epoch-based forward privacy is not inherently weaker than standard forward privacy, as the length of the epoch interval can be adjusted to enhance security, similar to epoch-based secure messaging systems

[32]. A shorter epoch interval results in stronger forward privacy, as the issued search tokens become invalid sooner.

4. Hidden ID-Coupling KAE

As discussed in §2.2, ICKAE leaks the embedded identity from the decryption key, leading to KGA. In this section, we introduce a new *Hidden*-ID Coupling KAE (HICKAE) scheme, which conceals the identity and offers the same properties as ICKAE (e.g., key-aggregate, anonymity). We start by giving its formal definitions.

4.1. Definitions

Definition 7 (HICKAE). *A* HICKAE *scheme is a tuple of PPT algorithms defined as follows:*

- $(\hat{\sigma}, \mathsf{pp}) \leftarrow \mathsf{Setup}(1^\lambda, n)$*: Given a security parameter* λ *and a number of classes* n*, it outputs a trapdoor* $\hat{\sigma}$ *and public parameters* pp.
- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$*: Given a security parameter* λ*, it outputs a public and private key pair* (pk, sk).
- $(\sigma'_i, \mathsf{ek}_i) \leftarrow \mathsf{IGen}(1^\lambda, i, \mathsf{pp})$*: Given a security parameter* λ*, a class identifier* i*, and public parameters* pp*, it outputs a class secret* σ'_i *and a class-binding key* ek_i.
- $(\Sigma, \mathsf{Corr}) \leftarrow \mathsf{Prep}(\mathsf{sk}, \hat{\sigma}, \sigma')$*: Given the private key* sk*, a trapdoor* $\hat{\sigma}$*, and classes' secret* $\sigma' = \{\sigma'_i\}_{i\in[n]}$*, it outputs a set of shared secrets* Σ*, and a set* Corr *that contains correlations between any two arbitrary classes* $i \neq j \in [n]$.
- $c \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{ek}_i, \mathsf{id}, m)$*: Given the public key* pk*, a class-binding key* ek_i*, an embedded identity* id*, and a plaintext* m*, it outputs the ciphertext* c *of* m.
- $\mathsf{ak} \leftarrow \mathsf{Ext}(\mathsf{sk}, S, \Sigma, \mathsf{id})$*: Given the private key* sk*, a set* $S \subseteq [n]$*, a shared secrets set* Σ*, and an embedded identity* id*, it outputs an aggregated key* ak.
- $m \leftarrow \mathsf{Dec}(\mathsf{ak}, S, i, c, \mathsf{Corr})$*: Given an aggregated key* ak*, a class set* $S \subseteq [n]$*, a class identifier* i*, a ciphertext* c*, and a correlation set* Corr*, it outputs plaintext* m.

Correctness. For any integers λ, n, any $S \subseteq [n]$, $i \in S$, id, and m, $\Pr[\mathsf{Dec}(\mathsf{ak}, S, i, c, \mathsf{Corr}) = m \mid (\hat{\sigma}, \mathsf{pp}) \leftarrow \mathsf{Setup}(1^\lambda, n), (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda), (\sigma'_i, \mathsf{ek}_i) \leftarrow \mathsf{IGen}(1^\lambda, i, \mathsf{pp}), (\Sigma, \mathsf{Corr}) \leftarrow \mathsf{Prep}(\mathsf{sk}, \hat{\sigma}, \sigma'), c \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{ek}_i, \mathsf{id}, m), \mathsf{ak} \leftarrow \mathsf{Ext}(\mathsf{sk}, S, \Sigma, \mathsf{id})] = 1$ with $\sigma' = \{\sigma'_i\}_{i\in[n]}$.

Compactness. The size of both the ciphertext and the aggregated key is independent of the number of classes.

Confidentiality, Anonymity and Aggregated Key with Hidden ID. In the chosen-plaintext attack game (Figure 2, IND-CPA), the adversary distinguishes a ciphertext of one of its chosen messages (i.e., m_0, m_1) under its specified class identifier (i.e., i^*). The anonymity game (IND-ANON) challenges the adversary with the ciphertext of its chosen message under one of two IDs (i.e., id^*_0, id^*_1) it specifies from the same class (i.e., i^*). The chosen-identity attack game (IND-CIA) challenges the adversary with the aggregated key under one of two IDs also from the same class.

Definition 8. HICKAE *is* X*-secure if for any* PPT $\mathcal{A}$,
$\left|\Pr[\mathsf{X}^0_{\mathsf{HICKAE},\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathsf{X}^1_{\mathsf{HICKAE},\mathcal{A}}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$

$\underline{\mathsf{Setup}(1^\lambda, n)}$:
1: $\alpha \xleftarrow{\$} \mathbb{Z}_q$, $\hat{\sigma} \xleftarrow{\$} \mathbb{Z}_p$
2: **for** $i \in [n]$ **do**
3: $\quad x_i \leftarrow \hat{\sigma}^i \pmod p$; $[p_i]_2 \leftarrow [\alpha^{-x_i}]_2$
4: **return** $(\hat{\sigma}, \mathsf{pp})$, where $\mathsf{pp} \leftarrow \{[p_i]_2\}_{i\in[n]}$

$\underline{\mathsf{KeyGen}(1^\lambda)}$:
5: $\tau \xleftarrow{\$} \mathbb{Z}_p$; $\gamma, \delta, \xi \xleftarrow{\$} \mathbb{Z}_q$
6: $\mathsf{pk} \leftarrow ([\gamma]_2, [\delta]_2, [\xi]_2)$; $\mathsf{sk} \leftarrow (\tau, \gamma, \delta, \xi)$
7: **return** $(\mathsf{pk}, \mathsf{sk})$

$\underline{\mathsf{IGen}(1^\lambda, i, \mathsf{pp})}$:
8: **parse** $\mathsf{pp} = \{[p_i]_2\}_{i\in[n]}$
9: $\sigma'_i \xleftarrow{\$} \mathbb{Z}_p$; $[\mathsf{ek}_i]_2 \leftarrow [p_i]_2 \alpha^{-\sigma'_i}$
10: **return** $(\sigma'_i, [\mathsf{ek}_i]_2)$

$\underline{\mathsf{Prep}(\mathsf{sk}, \hat{\sigma}, \sigma')}$:
11: **parse** $\mathsf{sk} = (\tau, \gamma, \delta, \xi)$, $\sigma' = \{\sigma'_i\}_{i\in[n]}$
12: **for** $i \in [n]$ **do** $\sigma_i \leftarrow \hat{\sigma}^i + \sigma'_i \pmod p$
13: **for** $i \neq j \in [n]$ **do**
14: $\quad [\Delta_{i,j}]_1 \leftarrow [\alpha^{\tau+\sigma_i-\sigma_j}]_1$; $[\Delta_{j,i}]_1 \leftarrow [\alpha^{\tau+\sigma_j-\sigma_i}]_1$
15: $\Sigma \leftarrow \{\sigma_i\}_{i\in[n]}$; $\mathsf{Corr} \leftarrow \{[\Delta_{i,j}]_1, [\Delta_{j,i}]_1\}_{i\neq j\in[n]}$
16: **return** (Σ, Corr)

$\underline{\mathsf{Enc}(\mathsf{pk}, [\mathsf{ek}_i]_2, \mathsf{id}, m)}$:
17: **parse** $\mathsf{pk} = ([\gamma]_2, [\delta]_2, [\xi]_2)$
18: $r \xleftarrow{\$} \mathbb{Z}_q$; $[c_1]_2 \leftarrow [r]_2$; $[c_2]_2 \leftarrow r[\xi]_2$; $[c_3]_2 \leftarrow r([\gamma]_2 + [\mathsf{ek}_i]_2)$
19: $[h]_1 \leftarrow H(\mathsf{id})$; $c_4 \leftarrow m \oplus G(r[h]_1[\delta]_2)$
20: **return** $c \leftarrow ([c_1]_2, [c_2]_2, [c_3]_2, c_4)$

$\underline{\mathsf{Ext}(\mathsf{sk}, S, \Sigma, \mathsf{id})}$:
21: **parse** $\mathsf{sk} = (\tau, \gamma, \delta, \xi)$, and $\Sigma = \{\sigma_i\}_{i\in[n]}$
22: $\tau' \xleftarrow{\$} \mathbb{Z}_p$; $k_1 \leftarrow \alpha^{\tau'} + \xi \pmod q$; $[h]_1 \leftarrow H(\mathsf{id})$
23: $[k_2]_1 \leftarrow \gamma \sum_{i\in S}[\alpha^{\tau+\sigma_i}]_1 + [\alpha^{\tau}]_1 + \delta[h]_1\alpha^{-\tau'}$
24: $[k_3]_1 \leftarrow \sum_{i\in S}[\alpha^{\tau+\tau'+\sigma_i}]_1$
25: **return** $\mathsf{ak} \leftarrow (k_1, [k_2]_1, [k_3]_1)$

$\underline{\mathsf{Dec}(\mathsf{ak}, S, i, c, \mathsf{Corr})}$:
26: **parse** $\mathsf{ak} = (k_1, [k_2]_1, [k_3]_1)$, and $c = ([c_1]_2, [c_2]_2, [c_3]_2, c_4)$
27: **parse** $\mathsf{Corr} = \{\{[\Delta_{j,k}]_1\}_{k\in[n]\setminus\{j\}}\}_{j\in[n]}$
28: $[u]_t \leftarrow ([k_2]_1 + \sum_{j\in S\setminus\{i\}}[\Delta_{j,i}]_1)([c_1]_2 k_1 - [c_2]_2) - [k_3]_1[c_3]_2$
29: **return** $c_4 \oplus G([u]_t)$

Figure 3: Our Proposed HICKAE scheme.

where the games $\mathsf{X}^b_{\mathsf{HICKAE},\mathcal{A}}(1^\lambda)$ *are defined in Figure 2 with* $\mathsf{X} \in \{\mathsf{IND\text{-}CPA}, \mathsf{IND\text{-}ANON}, \mathsf{IND\text{-}CIA}\}$.

4.2. Our Concrete HICKAE Scheme

We design HICKAE, an IBE-based KAE scheme that can hide the identity embedded in the decryption key. The idea is to bind secrets of encryptors (hereafter referred to as *classes*) into ciphertexts. These secrets prevent an untrusted third party (i.e., the server) that handles delegated decryption from brute-forcing all possible identities to learn which identity is currently bound to the provided decryption key. A class

secret is only known by the user, and each class owns a class-binding key computed from the secret of that class for encryption. The challenge is to enable decrypting all ciphertexts (associated with the same identity but different secrets) created by multiple classes using a single key. We resolve it by securely establishing correlations between two arbitrary classes.

Suppose the user holds a private key $\mathsf{sk} = (\tau, \gamma, \delta, \xi)$, where $\tau \xleftarrow{\$} \mathbb{Z}_p$, $\gamma, \delta, \xi \xleftarrow{\$} \mathbb{Z}_q$, publishes a public key $\mathsf{pk} = ([\gamma]_2, [\delta]_2, [\xi]_2)$ and public parameters $\mathsf{pp} \leftarrow \{[p_i]_2\}_{i\in[n]}$, where $[p_i]_2 \leftarrow [\alpha^{-x_i}]_2$ with $x_i \leftarrow \hat{\sigma}^i$ for $i \in [n]$, in which α is a generator of $\mathbb{Z}_q$ and $\hat{\sigma}$ is a secret trapdoor in $\mathbb{Z}_p$ only known by the user. Let $\sigma'_i \xleftarrow{\$} \mathbb{Z}_p$ be a secret of class i, for $i \in [n]$. Each class i computes its class-binding key $[\mathsf{ek}_i]_2 \leftarrow [p_i]_2\alpha^{-\sigma'_i} = [\alpha^{-\hat{\sigma}^i-\sigma'_i}]_2$, which is the input with the public key pk in encryption. The objective of Prep, which is executed by the user, is to obtain the shared secret of each class $i \in [n]$ as $\sigma_i \leftarrow \hat{\sigma}^i + \sigma'_i \in \mathbb{Z}_p$, and public correlation values $[\Delta_{i,j}]_1 = [\alpha^{\tau+\sigma_i-\sigma_j}]_1$ between two arbitrary classes $i, j \in [n]$ and $i \neq j$. In this setup process, the secret of classes σ_i, with $i \in [n]$, together with $[\alpha^\tau]_1$ and $[\alpha^\tau]_t$ are not revealed to ensure the security of confined decryption. The third-party uses the public correlations and the provided aggregated key given by the user for decryption later.

Let $H : \{0,1\}^* \to \mathbb{G}_1$ and $G : \mathbb{G}_t \to \{0,1\}^\lambda$ be two cryptographic hash functions. Figure 3 presents our HICKAE scheme in detail. Because each class is associated with a secret class-binding key, the adversary cannot create ciphertext under its chosen embedded identity itself to guess the queried one. Each aggregated key is confined to decrypt ciphertext of a class subset S, thus if a corrupt class is not included in S, the privacy of the queried identity is protected.

The size of an aggregated key does not depend on the number of target classes. When decrypting ciphertext of a target class i, the value $[\alpha^\tau]_1 r[\alpha^{\tau'}]_2 = r[\alpha^{\tau+\tau'}]_t$ that appears when evaluating $[k_2]_1([c_1]_2 k_1 - [c_2]_2)$ is canceled out by the value $[\alpha^{\tau+\tau'+\sigma_i}]_1 r[\alpha^{-\sigma_i}]_2 = r[\alpha^{\tau+\tau'}]_t$ when evaluating $[k_3]_1[c_3]_2$. For other classes not in the target subset, $r[\alpha^{\tau+\tau'}]_t$ cannot be canceled, which leads to failure in decryption, thereby preserving the message confidentiality of non-target classes. Also, $[\alpha^\tau]_1$ and $[\alpha^\tau]_t$ are secret, therefore $r[\alpha^{\tau+\tau'}]_t$ cannot be eliminated, which ensures the security of confined decryption. The correctness of HICKAE can be checked by noting that:

$$
\begin{aligned}
[u]_t &= ([k_2]_1 + \sum_{j\in S\setminus\{i\}} [\Delta_{j,i}]_1)([c_1]_2 k_1 - [c_2]_2) - [k_3]_1[c_3]_2 \\
&= \Big(\gamma \sum_{j\in S} [\alpha^{\tau+\sigma_j}]_1 + [\alpha^\tau]_1 + \delta[h]_1\alpha^{-\tau'} + \sum_{j\in S\setminus\{i\}} [\alpha^{\tau+\sigma_j-\sigma_i}]_1\Big) \\
&\times ([r]_2(\alpha^{\tau'} + \xi) - r[\xi]_2) - \sum_{j\in S} [\alpha^{\tau+\tau'+\sigma_j}]_1 r([\gamma]_2 + [\alpha^{-\sigma_i}]_2) \\
&= r[h]_1[\delta]_2
\end{aligned}
$$

Theorem 1. HICKAE *is* IND-CPA- *and* IND-ANON-*secure by Definition 8 under the BDH assumption (Definition 9, Appendix §A.1), and* IND-CIA-*secure by Definition 8 under the DL assumption (Definition 10, Appendix §A.1).*

We prove the IND-CPA and IND-ANON security of HICKAE by a reduction to the bilinear Diffie-Hellman (BDH) problem, i.e., compute $[hr\delta]_t$ from $([h]_1, [r]_2, [\delta]_2)$. We prove the IND-CIA security by a reduction to the discrete-log (DL) problem, i.e., compute $\alpha^{-\sigma_i}$ from $([1]_2, [\alpha^{-\sigma_i}]_2)$.

Proof. See details in Appendix §A.1. □

RSetup(1^λ):
1: $(\hat{\sigma}, \mathsf{pp}) \leftarrow \mathsf{HICKAE.Setup}(1^\lambda, n)$
2: $(\mathsf{rpk}, \mathsf{rsk}) \leftarrow \mathsf{HICKAE.KeyGen}(1^\lambda)$
3: **return** $(\mathsf{rpk}, \mathsf{rsk}, \hat{\sigma}, \mathsf{pp})$

WSetup$(1^\lambda, i, \mathsf{pp})$:
4: $(\sigma'_i, [\mathsf{ek}_i]_2) \leftarrow \mathsf{IGen}(1^\lambda, i, \mathsf{pp})$
5: $(\kappa_i, \mathsf{EIDX}_i) \leftarrow \mathsf{DSSE.Setup}(1^\lambda)$; $\mathsf{ESTkn}_i \leftarrow \{\varnothing\}$
6: $\mathsf{wsk}_i \leftarrow (\kappa_i, [\mathsf{ek}_i]_2)$
7: **return** $(\mathsf{wsk}_i, \sigma'_i, \mathsf{EIDX}_i, \mathsf{ESTkn}_i)$

RPrep$(\mathsf{rsk}, \hat{\sigma}, \sigma')$:
1: $(\Sigma, \mathsf{Corr}) \leftarrow \mathsf{HICKAE.Prep}(\mathsf{rsk}, \hat{\sigma}, \sigma')$
2: **return** (Σ, Corr)

Figure 4: Hermes Setup.

5. Our Proposed Hermes

We build Hermes from HICKAE to achieve multi-writer encrypted search with KGA resilience, sublinear search, and forward privacy. Note that in the multi-writer context, the writer is referred to as the encryptor/class in HICKAE.

5.1. Setup

We present Hermes setup procedure in Figure 4. Specifically, the reader executes RSetup procedure, which invokes the setup algorithm of HICKAE to generate a secret trapdoor $\hat{\sigma}$ and public parameters pp, followed by the key generation algorithm of HICKAE to generate a public/private key pair $(\mathsf{rpk}, \mathsf{rsk})$. Meanwhile, each writer i in the system executes WSetup procedure to generate a class secret σ'_i, obtain the class-binding key $[\mathsf{ek}_i]_2$ from its secret and the public parameters, also generates two main components: (*i*) the DSSE encrypted index EIDX_i initialized via DSSE.Setup; and (*ii*) the search access control component ESTkn_i containing a set of encrypted sharing tokens (WSetup, ln. 5). EIDX_i and ESTkn_i are initially empty. The writer keeps its class-binding key $[\mathsf{ek}_i]_2$ and the DSSE key κ_i secret, which will be employed to authorize keyword search ability to the reader and to update its DSSE index, respectively. Finally, for efficient search, the reader computes auxiliary information from its trapdoor and the writers' secrets, resulting in a set of shared secrets $\Sigma = \{\sigma_i\}_{i\in[n]}$ and correlation values Corr (Algorithm RPrep). The reader sends Corr to the server while keeping the shared secrets Σ private.

5.2. Update

We present the update procedure of Hermes in Figure 5, which permits a writer i to update (e.g., add/delete) a

keyword-file pair (w, f) in the index and authorize the reader to search. Specifically, writer i first generates a DSSE update token $\mathsf{u}_{\mathsf{sse}}$ using the back-end DSSE update procedure. To permit the reader to search for a keyword w and obtain the up-to-date result, the writer computes $\mathfrak{s}_{\mathsf{sse}}$, the latest DSSE trapdoor of w then encrypts it with HICKAE using the class-binding key $[\mathsf{ek}_i]_2$, in which w is treated as the embedded ID as $c_w \leftarrow \mathsf{HICKAE.Enc}(\mathsf{pk}, [\mathsf{ek}_i]_2, w, \mathfrak{s}_{\mathsf{sse}})$ (Figure 5, UpdtTkn, ln. 8). Since HICKAE achieves IND-ANON (i.e., ciphertext anonymity, see Theorem 1), c_w is guaranteed to conceal the embedded ID (i.e., keyword w).

5.2.1. Partitioning for Sublinear Search. As the encrypted sharing tokens are created with the reader's public key, prior work requires the server to match the reader's search token (created by its private key) with *all* sharing tokens, leading to linear search complexity. In Hermes, we introduce a novel partitioning strategy that can reduce the search complexity to sublinear while preserving KGA-resiliency. Let W be the keyword universe set. The idea is to let the writer cluster $|W|$ encrypted sharing tokens into $\sqrt{|W|}$ partitions so that when the reader executes a search, the server can identify which partition the keyword belongs to, and then test the reader's query against $\mathcal{O}(\sqrt{|W|})$ encrypted sharing tokens within that partition. The challenge lies in securely mapping an authorized keyword to a partition without being vulnerable to KGA. For example, with a public keyword-partition mapping H': $\{0,1\}^* \to [0, \sqrt{|W|} - 1]$, when the reader specifies the partition $H'(w)$ for efficient look-up during search, the server can execute dictionary attacks to learn what are the keywords that belong to $H'(w)$, thereby compromising keyword privacy.

To prevent the above attack, our idea is to map an authorized keyword to a randomized *physical* partition that is unlikable with the *logical* partition of the keyword. Specifically, given a keyword w, the writer first computes its logical partition as $\mathsf{pid} \leftarrow H'(w)$. The writer then computes its physical partition as $\mathsf{paddr} = F(\kappa, \mathsf{pid})$, where F: $\{0,1\}^\lambda \times [0, \sqrt{|W|} - 1] \to \{0,1\}^\lambda$ is a pseudorandom function (PRF) and κ is the writer's secret key. Finally, the writer encrypts the physical partition paddr with HICKAE using the logical partition pid as the embedded identity by $c_p \leftarrow \mathsf{HICKAE.Enc}(\mathsf{pk}, [\mathsf{ek}_i]_2, \mathsf{pid}, \mathsf{paddr})$ (Figure 5, ln. 5). When the reader searches, the server uses the delegated key to decrypt and obtain the physical partition of the queried keyword, and then only tests against encrypted tokens within that physical partition. Notice that it is completely safe to reveal the physical partition because it does not leak any information about the logical partition of the searched keyword due to the one-way property of PRF and IND-CIA security of HICKAE, thus achieving KGA-resiliency. Furthermore, it is necessary to compute the physical partition from the logical one instead of the keyword so that the number of ciphertext partitions can be bound to $\sqrt{|W|}$.

Let $\mathsf{ESTkn}_i = (\mathsf{PTkn}_i, \mathsf{WTkn}_i)$ be the set of encrypted sharing tokens of writer i, where PTkn_i stores encrypted partition tokens and WTkn_i stores encrypted keyword tokens. An updated keyword has two HICKAE ciphertexts including

$\mathsf{UpdtTkn}(\mathsf{rpk}, \mathsf{wsk}_i, i, \mathsf{op}, w, f)$:
1: **parse** $\mathsf{wsk}_i = (\kappa_i, [\mathsf{ek}_i]_2)$
2: $\mathsf{u}_{\mathsf{sse}} \leftarrow \mathsf{DSSE.UpdtTkn}(\kappa_i, \mathsf{op}, w, f)$
3: $\mathfrak{s}_{\mathsf{sse}} \leftarrow \mathsf{DSSE.SrchTkn}(\kappa_i, w)$
4: $\mathsf{pid} \leftarrow H'(w)$; $\mathsf{paddr} \leftarrow F(\kappa_i, \mathsf{pid})$
5: $c_p \leftarrow \mathsf{HICKAE.Enc}(\mathsf{rpk}, [\mathsf{ek}_i]_2, \mathsf{pid}, \mathsf{paddr})$
6: Let e be current epoch and $\mathbf{t}(e)$ be its encoding by Eq. (1)
7: **foreach** $\mathbf{t}' \in \Gamma_{\mathbf{t}(e)}$ ▷ See Eq. (2) for $\Gamma_{\mathbf{t}(e)}$
8: $\quad c_{w,\mathbf{t}'} \leftarrow \mathsf{HICKAE.Enc}(\mathsf{pk}, [\mathsf{ek}_i]_2, w \| \mathbf{t}', \mathfrak{s}_{\mathsf{sse}})$
9: **return** $\mathsf{u}_i \leftarrow (\mathsf{u}_{\mathsf{sse}}, \mathsf{paddr}, c_p, c_w \leftarrow \{c_{w,\mathbf{t}'}\}_{\mathbf{t}' \in \Gamma_{\mathbf{t}(e)}})$

$\mathsf{Updt}(\mathsf{u}_i, \mathsf{EIDX}_i, \mathsf{ESTkn}_i)$:
10: **parse** $\mathsf{u}_i = (\mathsf{u}_{\mathsf{sse}}, \mathsf{paddr}, c_p, c_w)$
11: **parse** $\mathsf{ESTkn}_i = (\mathsf{PTkn}_i, \mathsf{WTkn}_i)$
12: **if** $\mathsf{WTkn}_i[\mathsf{paddr}] = \{\varnothing\}$ **then** $\mathsf{PTkn}'_i \leftarrow \mathsf{PTkn}_i \cup \{c_p\}$
13: $\mathsf{WTkn}'_i[\mathsf{paddr}] \leftarrow \mathsf{WTkn}_i[\mathsf{paddr}] \cup \{c_w\}$
14: $\mathsf{EIDX}'_i \leftarrow \mathsf{DSSE.Updt}(\mathsf{u}_{\mathsf{sse}}, \mathsf{EIDX}_i)$
15: $\mathsf{ESTkn}'_i \leftarrow (\mathsf{PTkn}'_i, \mathsf{WTkn}'_i)$
16: **return** $(\mathsf{EIDX}'_i, \mathsf{ESTkn}'_i)$

Figure 5: Hermes Update.

the encrypted keyword token c_w and the encrypted partition token c_p. When the writer updates, it sends the DSSE update token $\mathsf{u}_{\mathsf{sse}}$, the partition address paddr, and the HICKAE ciphertext pair (c_p, c_w) to the server. The server appends c_p to the encrypted partition token set PTkn_i if paddr has not appeared previously (i.e., $\mathsf{WTkn}_i[\mathsf{paddr}]$ is empty), and appends c_w to the encrypted keyword sharing token set $\mathsf{WTkn}_i[\mathsf{paddr}]$ in the search access control of the writer (Figure 5, ln. 12–13). Finally, the server executes the standard DSSE update procedure (DSSE.Updt) on the DSSE component with $\mathsf{u}_{\mathsf{sse}}$ to update EIDX_i (Figure 5, ln. 14).

5.2.2. Forward Privacy without Rebuild. We present an efficient method that can achieve epoch-based forward privacy *without* requiring the writer to rebuild all sharing tokens periodically per epoch as in prior work [70].

Encoding of Epochs. To avoid a costly rebuild, our idea is to encode epoch values such that the aggregated keys provided for the decryption of current ciphertexts do not compromise the privacy of future updates, while are still valid for all ciphertexts up to the current epoch. To do so, our approach is to represent each epoch value as a unique set such that some of its elements appear in the set of the previous epoch, but not in the next epoch. The challenge is to minimize the cardinality of the set, thereby reducing the size of the aggregated decryption key. We adapt and refine the set representation in [31], [62] to ensure the cardinality of the set is only logarithmic in the number of epochs that can be supported as follows.

Suppose the epoch e corresponds to the time in the interval $[2^{\lambda+1} - 1]$ for some $\lambda \geq 0$. We construct a binary tree of height λ with $2^{\lambda+1} - 1$ nodes, each corresponding to the time in the interval $[2^{\lambda+1} - 1]$. Each node of the tree is represented by a string in $\{1,2\}^{\leq\lambda}$, where 1 denotes taking the left branch and 2 denotes taking the right branch. The bijection mapping tag $\mathbf{t} = (t_1, \dots) \in \{1,2\}^{\leq\lambda}$ to epoch e

is given by:

$$e(\mathbf{t}) = 1 + \sum_{i=1}^{|\mathbf{t}|} \Big(1 + (2^{\lambda+1-i} - 1)(t_i - 1)\Big)$$

For instance, with $\lambda = 2$, this maps the string $\epsilon, 1, 11, 12, 2, 21, 22$ to the epoch $1, 2, 3, 4, 5, 6, 7$, respectively, where ϵ is an empty string. We define an inverse function $\mathbf{t}$ to map epoch e to tag $\mathbf{t} \in \{1,2\}^{\leq\lambda}$ as:

$$\mathbf{t}(e) = \begin{cases} \epsilon & \text{if } e = 1 \\ \mathbf{t}(e-1)\|1 & \text{if } |\mathbf{t}(e-1)| < \lambda \\ \bar{\mathbf{t}}\|2 & \text{if } |\mathbf{t}(e-1)| = \lambda \end{cases} \quad (1)$$

in which $\bar{\mathbf{t}}$ is the longest string such that $\bar{\mathbf{t}}\|1$ is a prefix of $\mathbf{t}(e-1)$. The bijection induces a precedence relation over $\{1,2\}^{\leq\lambda}$, where $\mathbf{t} \preceq \mathbf{t}'$ (or $\mathbf{t}' \succeq \mathbf{t}$) iff either $\mathbf{t}$ is a prefix of $\mathbf{t}'$ or there exists $\bar{\mathbf{t}}$ such that $\bar{\mathbf{t}}\|1$ is a prefix of $\mathbf{t}$ and $\bar{\mathbf{t}}\|2$ is a prefix of $\mathbf{t}'$. Furthermore, any $\mathbf{t} \in \{1,2\}^{\leq\lambda}$ is associated with a set $\Gamma_{\mathbf{t}} \subset \{1,2\}^{\leq\lambda}$ given by:

$$\Gamma_{\mathbf{t}} = \{\mathbf{t}\} \cup \{\bar{\mathbf{t}}\|2 : \bar{\mathbf{t}}\|1 \text{ is a prefix of } \mathbf{t}\} \quad (2)$$

that corresponds to the set containing $\mathbf{t}$ and all the right-hand siblings of nodes on the path from $\mathbf{t}$ to the root, which also happens to be the smallest set of nodes that includes a prefix of all $\mathbf{t}' \succeq \mathbf{t}$.

Example 1. With $\lambda = 2$, there are $2^{\lambda+1} - 1 = 7$ epochs each is associated with the following set:

$$\Gamma_\epsilon = \{\epsilon\}; \Gamma_1 = \{1,2\}; \Gamma_{11} = \{11,12,2\}; \Gamma_{12} = \{12,2\};$$
$$\Gamma_2 = \{2\}; \Gamma_{21} = \{21,22\}; \Gamma_{22} = \{22\}$$

The sets $\Gamma_{\mathbf{t}}$ satisfy the following properties: (1) $\mathbf{t} \preceq \mathbf{t}' \Leftrightarrow \exists \mathbf{u} \in \Gamma_{\mathbf{t}}$ such that $\mathbf{u}$ is a prefix of $\mathbf{t}'$; (2) $\forall \mathbf{t}$, we have $\Gamma_{\mathbf{t}(e(\mathbf{t})+1)} = \Gamma_{\mathbf{t}} \setminus \{\mathbf{t}\}$ if $|\mathbf{t}| = \lambda$ or $\Gamma_{\mathbf{t}(e(\mathbf{t})+1)} = (\Gamma_{\mathbf{t}} \setminus \{\mathbf{t}\}) \cup \{\mathbf{t}\|1, \mathbf{t}\|2\}$ otherwise; (3) $\forall \mathbf{t}' \succeq \mathbf{t}$, we have $\forall \mathbf{u}' \in \Gamma_{\mathbf{t}'}$, $\exists \mathbf{u} \in \Gamma_{\mathbf{t}}$ such that $\mathbf{u}$ is a prefix of $\mathbf{u}'$. Note that every tag is of length exactly λ, which can be obtained by padding with zeros if necessary.

Given that the epoch value can be represented by a small unique set, the writer can embed the epoch time into the encrypted sharing tokens to achieve epoch-based forward-privacy. Specifically, the writer forms the embedded ID in the encrypted sharing token of keyword w as $\mathsf{id} = w\|\mathbf{t}$, where $\mathbf{t} \in \Gamma_{\mathbf{t}(e)}$ (Figure 5, ln. 6–8). Because $|\Gamma_{\mathbf{t}(e)}| \leq \lambda + 1$, the ciphertext is expanded by a factor of $\mathcal{O}(\lambda)$. Note that this method avoids synchronizing ciphertexts to the current epoch, making them valid for new decryption keys, as commonly done in forward-secure updatable encryption [35], [62].

5.3. Search

We present the search protocol of Hermes in Figure 6. To search for a keyword w on a writer subset S, the reader executes HICKAE.Ext on w and its logical partition $H'(w)$ to extract the decryption keys $\mathfrak{s}_w$ and $\mathfrak{s}_p$, respectively. As HICKAE achieves IND-CIA (see Theorem 1), $\mathfrak{s}_w$ and $\mathfrak{s}_p$ conceal w and its logical address $H'(w)$, respectively, and therefore, it offers resiliency against KGA. Remark that the encrypted sharing tokens in Hermes is forward-private due to the embedded encoded epoch. Therefore, to ensure that $\mathfrak{s}_w$ can decrypt corresponding encrypted sharing tokens created by a writer up to the current epoch e during a search, $\mathfrak{s}_w$

SrchTkn(rsk, S, Σ, w):
3: pid $\leftarrow H'(w)$; $\mathfrak{s}_p \leftarrow$ HICKAE.Ext(rsk, S, Σ, pid)
4: $P_e \leftarrow \{\mathbf{t}' : \mathbf{t}'$ is a prefix of $\mathbf{t}(e)\}$
5: **foreach** $\mathbf{t}' \in P_e$
6: $\quad \mathfrak{s}_{w,\mathbf{t}'} \leftarrow$ HICKAE.Ext(rsk, $S, \Sigma, w\|\mathbf{t}'$)
7: **return** $\mathfrak{s} \leftarrow (\mathfrak{s}_p, \mathfrak{s}_w \leftarrow \{\mathfrak{s}_{w,\mathbf{t}'}\}_{\mathbf{t}' \in P_e})$

Srch($\mathfrak{s}, S$, EIDX, ESTkn, Corr):
8: **parse** $\mathfrak{s} = (\mathfrak{s}_p, \mathfrak{s}_w = \{\mathfrak{s}_{w,\mathbf{t}'}\}_{\mathbf{t}' \in P_e})$
9: **parse** EIDX $= \{\mathsf{EIDX}_i\}_{i\in[n]}$, and ESTkn $= \{\mathsf{ESTkn}_i\}_{i\in[n]}$
10: **foreach** $i \in S$
11: $\quad$ **parse** $\mathsf{ESTkn}_i = (\mathsf{PTkn}_i, \mathsf{WTkn}_i)$
12: $\quad$ **foreach** $c_p \in \mathsf{PTkn}_i$
13: $\quad\quad$ paddr $\leftarrow$ HICKAE.Dec($\mathfrak{s}_p, S, i, c_p$, Corr)
14: $\quad\quad$ **if** paddr $\neq \perp$ **then**
15: $\quad\quad\quad$ **foreach** $c_{w,\mathbf{t}'} \in \mathsf{WTkn}_i[\text{paddr}] : \mathbf{t}' \in P_e$
16: $\quad\quad\quad\quad \mathfrak{s}_{\mathsf{sse}} \leftarrow$ HICKAE.Dec($\mathfrak{s}_{w,\mathbf{t}'}, S, i, c_{w,\mathbf{t}'}$, Corr)
17: $\quad\quad\quad\quad$ **if** $\mathfrak{s}_{\mathsf{sse}} \neq \perp$ **then**
18: $\quad\quad\quad\quad\quad \mathcal{R}_i \leftarrow$ DSSE.Srch($\mathfrak{s}_{\mathsf{sse}}$, EIDX$_i$)
19: **return** $\mathcal{R} \leftarrow \cup_i \mathcal{R}_i$

Figure 6: Hermes Search.

must be created by associating w with the set P_e of prefixes of the encoded epoch $\mathbf{t}(e)$ (Figure 6, ln. 4–6). Following the **Example 1** above, at epoch $e = 4$, with $\mathbf{t}(e) = \mathbf{t}(4) = 12$ (Eq. (1)), the reader creates an aggregated decryption key with $\mathbf{t} \in P_4 = \{12, 1, \epsilon\}$, which can decrypt all ciphertexts up to epoch $e = 4$ but not for those from epochs $e \geq 5$. As the prefix set cardinality is $\leq \lambda + 1$, the decryption key is expanded by a factor of $\mathcal{O}(\lambda)$. To this end, the reader sends the search token $\mathfrak{s} = (\mathfrak{s}_p, \mathfrak{s}_w)$ to the server.

Next, for each writer $i \in S$, the server uses $\mathfrak{s}_p$ to decrypt encrypted partition tokens in PTkn_i, and obtain the physical partition address paddr of w. Then, the server uses $\mathfrak{s}_w$ to decrypt the authorized keyword sharing tokens in WTkn_i[paddr], thereby obtaining a DSSE trapdoor $\mathfrak{s}_{\mathsf{sse}}$. With $\mathfrak{s}_{\mathsf{sse}}$, the server executes the standard DSSE search protocol DSSE.Srch over the corresponding EIDX_i and obtains the search result $\mathcal{R}_i$. Finally, the server returns the search result $\mathcal{R} = \cup_i \mathcal{R}_i$ to the reader, which contains all file identifiers that match the queried keyword. As sharing tokens are uniformly distributed into $\sqrt{|W|}$ physical partitions, and each partition contains $\mathcal{O}(\sqrt{|W|})$ unique tokens, the expected computational search complexity is $\mathcal{O}(\lambda\sqrt{|W|})$.

5.4. Optimizations

We present several techniques that can be further applied to optimize search and update efficiency in Hermes.

Preventing Ever-Growing Search Access Control List. When an update on a keyword w happens, a new HICKAE ciphertext is appended into a partition WTkn[paddr] to ensure the reader could always get the latest results. This leads to an issue that there might be a lot of encrypted sharing tokens corresponding to the same keyword in a partition of the sharing component, which enlarges the search access control component over time and leads to

increased search complexity. Therefore, it is necessary to remove stale keyword-sharing tokens and only retain the latest ones. To address this issue, we can instantiate each partition $\mathsf{WTkn}[\mathsf{paddr}]$ as a stack, where every newly updated encrypted keyword-sharing token is placed on the top of the stack. During search, the server can prioritize decrypting the ciphertexts on top of the stack and return the first match. To remove outdated ciphertexts, the server can continue decrypting the remaining ciphertexts in $\mathsf{WTkn}[\mathsf{paddr}]$ to find other matches that are stale tokens and eliminate them. By doing this, the size of partitions in the keyword-sharing set WTkn will not be augmented due to dynamic updates and the search complexity can be maintained.

Further Reducing Search Complexity. In Hermes, the search complexity is $\mathcal{O}(\lambda\sqrt{|W|})$ via partitioning. In fact, we can reduce this complexity further by applying the proposed partitioning strategy recursively, where $\mathcal{O}(\sqrt{W})$ smaller physical partitions are treated as separate databases to continue clustering. Let $|W'| = \mathcal{O}(\sqrt{|W|})$ be the size of each partition after the first partitioning step. With one more level of partitioning, we can reduce the partition size to $\mathcal{O}(\sqrt{|W'|}) = \mathcal{O}(\sqrt[4]{|W|})$. Similarly, if we apply this strategy recursively $\frac{\log|W|}{\log\log|W|}$ times, the size of each partition becomes $\mathcal{O}(2^{\log\log|W|}) = \mathcal{O}(\log|W|)$ because $|W| = 2^{\log|W|} = 2^{\log\log|W|\frac{\log|W|}{\log\log|W|}}$. Thus, the computational search complexity on the server is $\mathcal{O}(\frac{\log^2|W|}{\log\log|W|})$ with a larger search token size since we need one more token for each partitioning level, which is $\mathcal{O}(\frac{\lambda\log|W|}{\log\log|W|})$ as a trade-off.

5.5. Analysis

Complexity. A Hermes search traverses the target token sets, which costs $\mathcal{O}(\lambda\sqrt{|W|})$ for each collection, where W is set of active keywords encrypted by the target writer, before executing a DSSE search with complexity $\mathcal{O}(d_w)$, where d_w is the number of updates on the queried keyword. Server computation is linear in the writer subset size, but search query size is constant (i.e., $\mathcal{O}(\lambda^2)$) thanks to the compactness of the HICKAE decryption key, which is independent of the number of databases to be searched. For each keyword-file pair update, the token size is $\mathcal{O}(\lambda^2)$ and the time complexity to create an update token is $\mathcal{O}(\lambda)$. If using the recursive partitioning technique, the server search complexity of Hermes is reduced to $\mathcal{O}(\log^2|W|/\log\log|W|+d_w)$ at the cost of larger search token size, which is $\mathcal{O}(\lambda\log|W|/\log\log|W| + \lambda^2)$ as a trade-off. Our scheme thus realizes sublinear search and also achieves writer efficiency. No direct interaction between the reader and any writer during search and update is needed.

Security. We state the security of Hermes as follows.

Theorem 2. Hermes *is* $\mathcal{L}_{\mathsf{mse}}$*-adaptively-secure with forward privacy if* HICKAE *is* IND-CPA-, IND-ANON- *and* IND-CIA-*secure, and* DSSE *is* $\mathcal{L}_{\mathsf{sse}}$*-adaptively-secure with forward privacy, where* $\mathcal{L}^{\mathsf{Setup}}_{\mathsf{mse}}(1^\lambda) = \{i\}_{i\in[n]}$, $\mathcal{L}^{\mathsf{Crpt}}_{\mathsf{mse}}(i) = \{\mathsf{UpdtBy}(i)\}$,

$$\mathcal{L}^{\mathsf{Srch}}_{\mathsf{mse}}(w,S) = \begin{cases} \{i, \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w), w\}_{i\in S} & \text{if } i \in \mathcal{I}_c, \\ \{i, \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w)\}_{i\in S} & \text{otherwise.} \end{cases}$$

and $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{mse}}(i,\mathsf{op},w,f) = \begin{cases} \{i,\mathsf{op},w,f\} & \text{if } i \in \mathcal{I}_c, \\ \{i, \mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}(\mathsf{op},w,f)\} & \text{otherwise.} \end{cases}$

Proof. See Appendix §A.2. □

6. Experiment

Implementation. We fully implemented all our proposed techniques in C++ consisting of approximately 2,000 lines of code. We used standard cryptographic libraries, including OpenSSL [6] for PRF and hash functions, PBC [7] and GMP library [4] for implementing pairing operations and arithmetic computations, respectively. We used libzeromq [8] to implement network communication between server and client. Our source code is available at: https://github.com/vt-asaplab/Hermes.

Hardware and network. We used a server with 8-core CPU @ 3.6 GHz and 64 GB memory. For reader/writers, we used a laptop with CPU @ 2.7 GHz and 16 GB RAM as a client. The network bandwidth between server and client is 30 Mbps with 7ms round-trip latency.

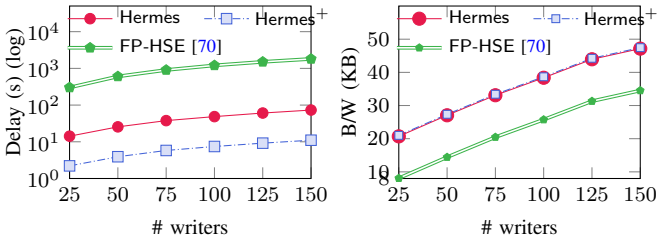
Dataset. We evaluated Hermes schemes over three real-world multi-writer datasets used by prior work [70].

• *Enron Email Dataset* [11]: It includes about 500K emails of 150 employees. We leveraged the same standard tokenization method described in Oblix [55]. We stemmed the words and removed stopwords and words that were > 20 or < 4 characters long or contained non-alphabetic characters. We considered each employee as a separate writer. Overall, each collection has an average of 10,944 keywords with a standard deviation of 7,017.

• *Diabetes Dataset (EHR)* [12]: 130 US hospitals contribute 101,766 patient records. For each record, we consider the combination of sex (binary) and age interval (10 years each) as its keyword and the rest as its data.

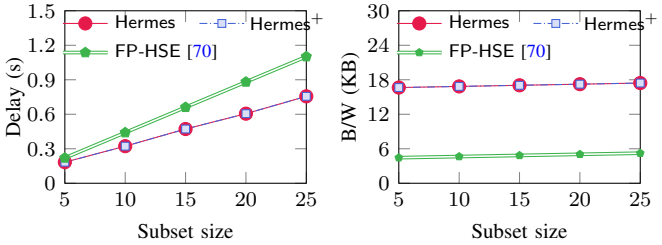
• *Room Climate Dataset (Sensor)* [13]: It contains 540,364 records of human activities under continuous measurements of room climate information, collected by 12 IoT sensors located in different places. We consider the climate data (i.e., temperature and relative humidity) as keywords after rounded up to the nearest integer.

Counterparts and Parameters Selection. For evaluation, we created two instantiations of our scheme: (*i*) Hermes with HICKAE, one-level of partitioning, and forward privacy without rebuild described in §5.2.2; and (*ii*) Hermes$^+$ is the same as Hermes yet the search complexity is reduced via recursive partitioning described in §5.4. We compared our Hermes schemes with FP-HSE [70], the state-of-the-art multi-writer SE scheme that has the same system model (i.e., single-server) as Hermes. We selected its parameters similar to ours. In particular, we used 256-bit keys for IND-CPA encryption and PRFs. We used SHA-512 for hash function



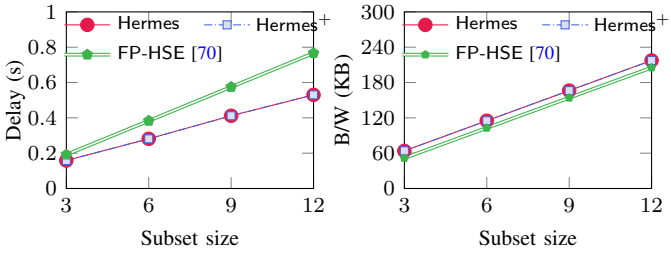
(a) E2E delay **(b)** Reader's bandwidth

Figure 7: E2E keyword search delay and bandwidth (Enron).



(a) E2E delay **(b)** Reader's bandwidth

Figure 8: E2E keyword search delay and bandwidth (EHR).



(a) E2E delay **(b)** Reader's bandwidth

Figure 9: E2E keyword search delay and bandwidth (Sensor).

and MNT224 curve for pairings with a 96-bit security level. For Hermes/Hermes$^+$, we chose $\lambda = 63$ as the length of encoded epochs to allow up to $2^{64} - 1$ epochs.

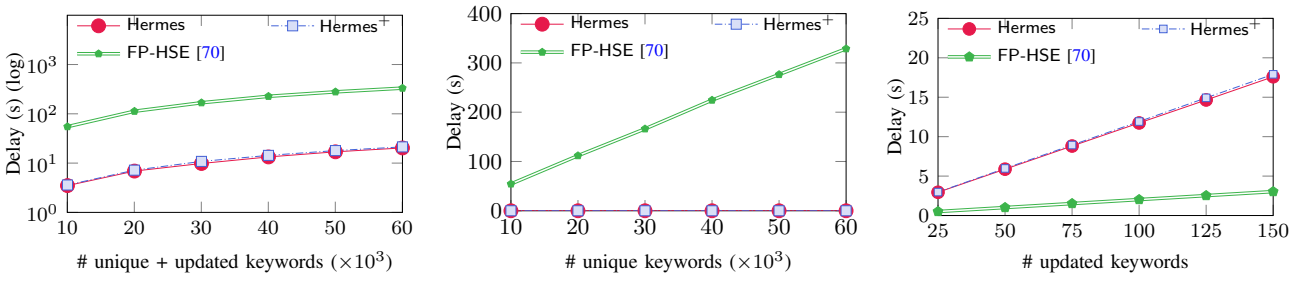
6.1. Overall Results

Keyword search. Figure 7a illustrates the end-to-end delay in keyword search of our schemes and FP-HSE for Enron dataset with different numbers of writers. The latency of all grows almost linearly in the number of writers. Hermes is about 21.2×–24.7× faster than FP-HSE, while Hermes$^+$ is 136.5×–163.8× faster than FP-HSE. With 25 writers, Hermes and Hermes$^+$ take approximately 14.2s and 2.2s, respectively, to process a search, and increase to about 73.2s and 11.0s, respectively, for 150 writers, while FP-HSE takes about 301.3s–1807.9s. The overhead of keyword search mainly comes from pairing operations on the server, in which decrypting each encrypted sharing token in the search access control component needs two pairing operations. While Hermes and Hermes$^+$ only traverse a sublinear number of encrypted tokens, FP-HSE has to check for all active keywords, incurring significant longer latency. For Figure 8a–9a, since EHR and Sensor only contain a small number of keywords, the performance gap between Hermes schemes and FP-HSE is closer, in which Hermes and Hermes$^+$ outperform FP-HSE by 1.2×–1.5× for EHR and 1.2×–1.4× for Sensor.

Reader's bandwidth. Figure 7b shows the search bandwidth between the reader and the server for our Hermes schemes and FP-HSE on the Enron dataset. The network overhead in Hermes and Hermes$^+$ increases from 20.7 KB to 47.2 KB and 21.0 KB to 47.5 KB, respectively, corresponding to the cases ranging from 25 to 150 writers, in which most is for receiving search output, while that of FP-HSE is 8.1 KB–34.5 KB. For Hermes and Hermes$^+$, they incur 12.6 KB–12.9 KB more network overhead per search operation than FP-HSE as its search query aggregates decryption power for all ciphertexts up to the current epoch. Although Hermes and Hermes$^+$ require larger communication cost for search, they do not need costly rebuild to maintain forward privacy as FP-HSE. A similar tendency can be observed in Figure 8b–9b. For Figure 9b, the number of matching results in DSSE is large, which accounts for significant amount of communication overhead and dominates the query size. Thus, the bandwidth costs of all schemes are close together.

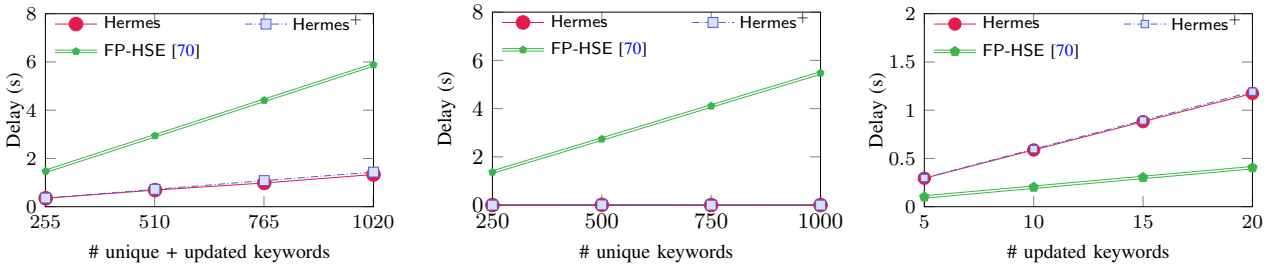
Keyword update. Figure 10a presents the end-to-end delay in keyword update of our Hermes schemes and FP-HSE for different database sizes based on the number of unique keywords and varying numbers of updates. For forward privacy, FP-HSE needs a rebuild to update all ciphertexts to the newly incremented epoch, followed by executing updates. We increase the size of keyword universe set from 10^4 to 6.10^4, and increase the number of updated keywords from 25 to 150. It is noticeable that when increasing the number of unique and updated keywords, the total latency of all grows linearly. For FP-HSE, its latency is around 54.9s–331.7s, while Hermes and Hermes$^+$ only take 3.5s–21.3s to process 25–150 keyword updates since they do not require a costly rebuild, thus is 15.3×–17.1× faster than FP-HSE. For smaller numbers of updated keywords, Hermes schemes are faster than FP-HSE 3.8×–4.2× for EHR (Figure 11a) and 1.8×–2.2× for Sensor (Figure 12a).

Figure 10b shows the separate rebuild cost w.r.t. varying keyword universe set sizes 10^4–6.10^4, where FP-HSE takes 54.4s–328.7s as the end-to-end delay, and Hermes and Hermes$^+$ do not require rebuild. Figure 11b–12b illustrate the rebuild latency for smaller numbers of keywords on EHR and Sensor datasets, respectively, which costs 0.8s–27.4s for FP-HSE. Figure 10c shows the update delay of Hermes schemes and FP-HSE (without rebuild) for different numbers of newly updated keywords. FP-HSE takes 0.5s-3.0s to process for the cases increasing from 25 to 150 keywords. For Hermes and Hermes$^+$, their latency is about 2.9s–17.9s to update the index and the search access control with forward privacy (without rebuild). Also, Hermes and Hermes$^+$ need to compute and enclose an encrypted partition token for each updated keyword. However, this cost can be reduced by storing the state of each partition on the writer to check if the partition is currently empty or not, and then only enclosing necessary encrypted partition tokens in updates. Figure 11c–12c demonstrate similar trends in the update



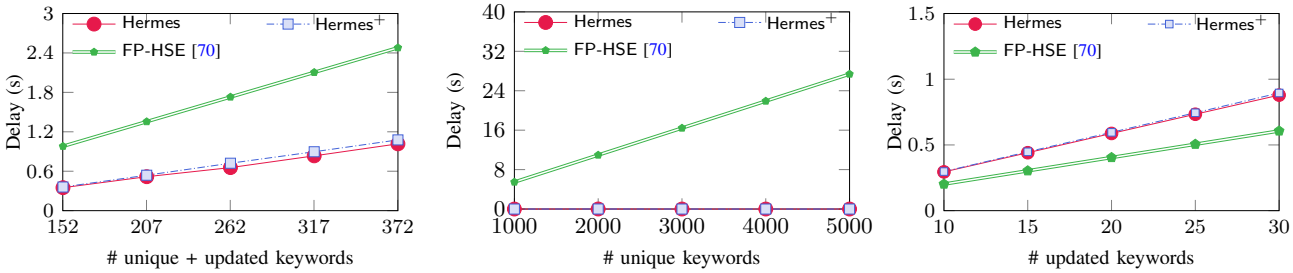
(a) Total end-to-end delay **(b)** Index rebuild cost **(c)** Index update cost

Figure 10: Keyword update delay (Enron). The total update cost with forward privacy (FP) **(a)** equals the cost to rebuild the index **(b)** plus the cost to update the index **(c)**. Unlike FP-HSE, Hermes/Hermes$^+$ does not need a rebuild process to ensure FP of subsequent updates.



(a) Total end-to-end delay **(b)** Index rebuild cost **(c)** Index update cost

Figure 11: Keyword update delay (EHR).



(a) Total end-to-end delay **(b)** Index rebuild cost **(c)** Index update cost

Figure 12: Keyword update delay (Sensor).

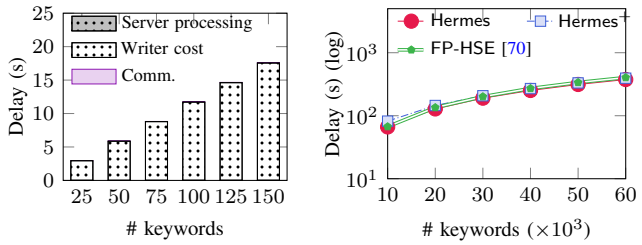
latency of Hermes schemes and FP-HSE but for smaller numbers of updated keywords.

6.2. Cost Breakdown

Keyword update. Since Hermes is close to Hermes$^+$ regarding the total update latency (as shown in Figure 10c), we only present the detailed keyword update cost of Hermes in Figure 13a. As most overhead is dominated by writer processing, the latency incurred by the server and communication is hardly visible. For 25–150 updated keywords, the server only takes 0.6ms–2.9ms, which is about 0.02% of the total delay to process updates based on update tokens received from the writer. The network time to send update tokens by the writer is 7.7ms–46.4ms, corresponding to 0.3% of the total delay. Most overhead is for writer processing in which it takes 2.9s–17.5s, which accounts for about 99.7% of the delay to create update tokens for 25-150 keywords.

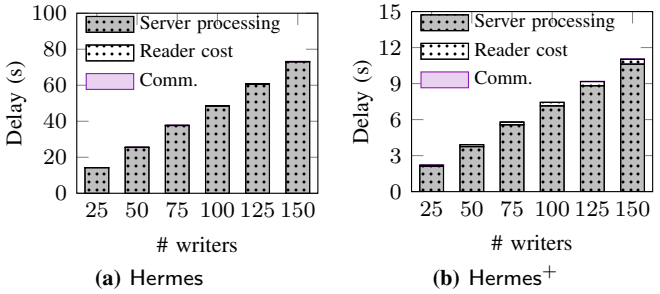
Keyword search. Figure 14a demonstrates the detailed cost of keyword search of Hermes for varying numbers of writers. Three factors contributing to the total delay include reader processing, communication latency and server processing. The reader takes 36.1ms–192.8ms, accounting for 0.3% of the total to create a search token, which is hardly visible since most latency is dominated by the overhead on the server. The communication delay is 9.3ms–16.5ms, accounting for less than 0.1% of the whole latency. Most overhead stems from server processing, where it takes 14.2s–72.9s, corresponding to 99.7% of the total delay to execute search for different subset sizes from 25 to 150 writers. We present the detailed cost of Hermes$^+$ in Figure 14b. The reader latency, communication and server overhead correspondingly are 98.4ms–409.2ms, 12.7ms–19.9ms, and 2.1s–10.6s, which contribute 3.7%–4.5%, 0.2%–0.6%, and 94.9%–96.1% to the total delay, respectively.

Setup time. Figure 13b shows the end-to-end setup time of Hermes and FP-HSE. Hermes, Hermes$^+$ and FP-HSE take 66.4s–375.4s, 82.0s–394.3s and 67.1s–403.5s, respectively,



(a) Hermes/Hermes$^+$ update **(b)** Setup time

Figure 13: Detailed cost of update and E2E setup time.



(a) Hermes **(b)** Hermes$^+$

Figure 14: Detailed cost of keyword search (Enron).

to initialize their search index and search access control corresponding to the number of unique keywords 10^4–6.10^4.

Storage Cost. In Hermes, the reader stores a 112 B private key and a 28 B shared secret per writer, which is 4.2 KB in total. Each writer stores a 168 B class-binding key and a 504 B public key. For Enron, each writer outsources an access control component of 62.8 MB on average, and a DSSE index of about 10.8 MB to the server, which is approximately 10.8 GB in total for 150 writers in the whole dataset.

6.3. Discussion

Hermes adapts the modular hybrid design from [70], which consists of two independent components: a DSSE-encrypted index and a search access control component. Consequently, to ensure a fair comparison with our main counterpart [70] and to highlight Hermes's advantages in providing more secure and efficient search access control, we implemented the DSSE-encrypted index in our experimental evaluation using a standard forward-private DSSE scheme similar to that in [70].

However, it is important to note that Hermes, like any HSE design, is not restricted to a specific DSSE scheme for constructing the encrypted index. In fact, the encrypted index in Hermes can be instantiated with various advanced DSSE designs, such as backward-private schemes [18], [65], volume-hiding mechanisms [46], [14], [14], [65], leakage-suppression techniques [47], [36], and search access pattern obfuscation [25], [64] for enhanced security. This flexibility is a key advantage of the hybrid SE model, making it more desirable than prior multi-user SE approaches. When Hermes is instantiated with these advanced DSSE schemes, the overall performance overhead comprise the experimental results we reported, plus any additional overhead specific to the chosen DSSE scheme for processing the encrypted index after obtaining the keyword trapdoor from the search access control component.

7. Related Work

(D)SSE. Song et al. [63] were the first to propose and formalize SE, which was later extended by a series of DSSE schemes that offer secure search over encrypted data plus dynamic update via an encrypted index [22], [37], [66], [30], [40], [23]. Early DSSE constructions, however, suffer from significant leakage (e.g., updated keywords, historical updates, volume), making them vulnerable to severe attacks (e.g., file-injection [77], leakage-abuse [21], [45], [73]). Substantial advancements have since been made to enhance the security of DSSE by enabling desirable security properties such as forward privacy [17], backward privacy [18], [65], volume-hiding [46], [14], [14], [65]. While DSSE leaks search and access patterns, attacks exploiting these patterns [58], [72], [38] require additional stringent assumptions, such as auxiliary knowledge of the user's exact query sequence and sparsely distributed queries, which may be difficult to achieve in practice [45]. Meanwhile, some approaches sought to obfuscate patterns leakage for added assurance, using cryptographic techniques such as Oblivious RAM or Private Information Retrieval [42], [29], [51], or differential privacy [61], leading to high computation and communication costs. DSSE mainly supports single-user functionalities, where the encrypted data can only be searched or updated by its owner.

Multi-User SE. Several approaches have been proposed to enable different multi-user encrypted search functionalities (e.g., multi-reader, multi-writer, or both [44]). Some extend DSSE to multi-reader encrypted search by sharing the search token of the data owner with multiple readers through key exchange [24], broadcast encryption [28], or proxy re-encryption [39]. Other works enable both multi-reader and multi-writer on top of DSSE by either requiring all users to be trusted [29] or harnessing distributed servers with secure computation to enforce search access control [51].

PKSE offers multi-writer encrypted search without requiring strong assumptions, where the writers can authorize the reader by encrypting their authorized keywords with the reader's public-key encryption [16], [75], [15], [74], [54], [52]. Some PKSE approaches leverage key-aggregate encryption to make the reader's query size more compact [27], [59], [71]. However, most PKSE designs face performance and security challenges including the lack of forward privacy [16], [15], [74], [52], [54], vulnerability to KGA [19], and high processing overhead due to linear search complexity and expensive public-key operations. Some attempt to prevent KGA by using a third party/authority [74], [71], [54], [52], distributed servers [49], [20], [53], or via public-key authenticated encryption [43] (yet numerous attacks were found [57], [60]). Recently, Wang et al. [70] proposed a new hybrid SE model that combines PKSE and DSSE to enable multi-writer encrypted search with sublinear search complexity (in the writer's database size). However, the

proposed hybrid design incurs a high overhead on the writer for forward privacy and is vulnerable to KGA. A recent hybrid scheme is resilient to KGA and offers forward privacy; however, it requires distributed servers, incurs linear search complexity, and does not offer compact search [50] due to the use of standard public-key encryption.

TEE-based SE. There are several SE systems that use a Trusted Execution Environment (TEE) (e.g., AMD SEV, Intel SGX) to enable encrypted query functionalities with diverse functionalities (e.g., SQL [33]) and security features (e.g., pattern obfuscation [41], [34], [55], forward and/or backward privacy [67], [68]). However, these systems require strong security assumptions on the security hardware (e.g., tamper-free, isolation, side-channel resistance).

8. Conclusion

In this paper, we proposed and designed Hermes, a new HSE scheme that offers multi-writer encrypted search functionalities with high-security guarantees and efficient performance. Compared with the prior multi-writer designs (e.g., HSE, PKSE), Hermes offers an optimized search complexity, which is sublinear in the number of active keywords. In addition, our scheme can prevent KGAs and maintain forward privacy with low writer overhead, which has become an essential security standard for searchable encrypted systems.

Acknowledgments

The authors would like to thank the shepherd and anonymous reviewers for their insightful comments and constructive feedback on improving the quality of this work. This work was supported in part by the Commonwealth Cyber Initiative (CCI) and by the National Science Foundation (NSF) under Grant No. 2350215.

References

[1] Barracuda Email Protection: Email security, made radically easy. https://www.barracuda.com/products/email-protection.

[2] Connecta Mobile: Secure. Private. Yours. Regain Control. Your Data. Your Rules. https://www.connectamobile.com/.

[3] Duality Tech: Secure, Privacy Protected Data Collaboration. https://dualitytech.com.

[4] GMP Library: The gnu multiple precision arithmetic library. https://gmplib.org/.

[5] Mastermost: Secure Collaboration Platform: Collaboration for Your Mission-Critical Work. https://mattermost.com.

[6] OpenSSL: Cryptography and SSL/TLS toolkit. https://www.openssl.org.

[7] PBC Library: The pairing-based cryptography library. https://crypto.stanford.edu/pbc/.

[8] ZeroMQ: An open-source universal messaging library. https://github.com/zeromq/libzmq.

[9] Sync: Secure cloud storage, privacy guaranteed. https://www.sync.com/, 2011.

[10] Tresorit: Secure file sharing & content collaboration with encryption. https://tresorit.com/secure-file-sharing, 2013.

[11] Enron dataset. https://www.cs.cmu.edu/~enron, 2015.

[12] Uc irvine diabetes dataset 1999-2008. https://github.com/jonneff/Diabetes2/, 2016.

[13] Room climate datasets. https://github.com/IoTsec/Room-Climate-Datasets/, 2017.

[14] Ghous Amjad, Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Dynamic volume-hiding encrypted multi-maps with applications to searchable encryption. *Proc. Priv. Enhancing Technol.*, 2023(1):417–436, 2023.

[15] Rouzbeh Behnia, Muslum Ozgur Ozmen, and Attila Altay Yavuz. Lattice-based public key searchable encryption from experimental perspectives. *IEEE Transactions on Dependable and Secure Computing*, 17(6):1269–1282, 2020.

[16] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. Cryptology ePrint Archive, Paper 2003/195, 2003. https://eprint.iacr.org/2003/195.

[17] Raphael Bost. $\sum o\phi o\varsigma$: Forward secure searchable encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1143–1154, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Raphael Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1465–1482, 2017.

[19] Jin Wook Byun, Hyun Suk Rhee, Hyun-A Park, and Dong Hoon Lee. Off-line keyword guessing attacks on recent keyword search schemes over encrypted data. In Willem Jonker and Milan Petković, editors, *Secure Data Management*, pages 75–83, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[20] Chengjun Cai, Yichen Zang, Cong Wang, Xiaohua Jia, and Qian Wang. Vizard: A metadata-hiding data analytic system with end-to-end policy controls. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 441–454, New York, NY, USA, 2022. Association for Computing Machinery.

[21] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 668–679, New York, NY, USA, 2015. Association for Computing Machinery.

[22] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Annual cryptology conference*, pages 353–373. Springer, 2013.

[23] Javad Ghareh Chamani, Dimitrios Papadopoulos, Mohammadamin Karbasforushan, and Ioannis Demertzis. Dynamic searchable encryption with optimal search in the presence of deletions. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022*, pages 2425–2442. USENIX Association, 2022.

[24] Javad Ghareh Chamani, Yun Wang, Dimitrios Papadopoulos, Mingyang Zhang, and Rasool Jalili. Multi-user dynamic searchable symmetric encryption with corrupted participants. *IEEE Transactions on Dependable and Secure Computing*, 20(1):114–130, 2023.

[25] Guoxing Chen, Ten-Hwang Lai, Michael K. Reiter, and Yinqian Zhang. Differentially private access patterns for searchable symmetric encryption. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 810–818, 2018.

[26] Cheng-Kang Chu, Sherman S.M. Chow, Wen-Guey Tzeng, Jianying Zhou, and Robert H. Deng. Key-aggregate cryptosystem for scalable data sharing in cloud storage. *IEEE Transactions on Parallel and Distributed Systems*, 25(2):468–477, 2014.

[27] Baojiang Cui, Zheli Liu, and Lingyu Wang. Key-aggregate searchable encryption (kase) for group data sharing via cloud storage. *IEEE Transactions on Computers*, 65(8):2374–2385, 2016.

[28] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 79–88, New York, NY, USA, 2006. Association for Computing Machinery.

[29] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. Dory: An encrypted search system with distributed trust. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020.

[30] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. 01 2020.

[31] Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. Pixel: Multi-signatures for consensus. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2093–2110. USENIX Association, August 2020.

[32] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1775–1792, August 2021.

[33] Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. *Proceedings of the VLDB Endowment*, 13(2), 2019.

[34] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. Hardidx: Practical and secure index with sgx. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 386–408. Springer, 2017.

[35] Yao Jiang Galteland and Jiaxin Pan. Backward-leak uni-directional updatable encryption from (homomorphic) public key encryption. In *International Conference on Theory and Practice of Public Key Cryptography*, 2023.

[36] Marilyn George, Seny Kamara, and Tarik Moataz. Structured encryption and dynamic leakage suppression. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 370–396, Cham, 2021. Springer International Publishing.

[37] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1038–1055, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Zichen Gui, Kenneth G. Paterson, and Sikhar Patranabis. Rethinking searchable symmetric encryption. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1401–1418, 2023.

[39] Ariel Hamlin, Abhi Shelat, Mor Weiss, and Daniel Wichs. Multi-key searchable encryption, revisited. In Michel Abdalla and Ricardo Dahab, editors, *Public-Key Cryptography – PKC 2018*, pages 95–124, Cham, 2018. Springer International Publishing.

[40] Thang Hoang, Ceyhun D. Ozkaptan, Gabriel Hackebeil, and Attila Altay Yavuz. Efficient oblivious data structures for database services on the cloud. *IEEE Transactions on Cloud Computing*, 9(2):598–609, 2021.

[41] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. Hardware-supported oram in effect: Practical oblivious search and update on very large dataset. *Proceedings on Privacy Enhancing Technologies*, 2019(1), 2019.

[42] Thang Hoang, Attila Yavuz, F. Durak, and Jorge Guajardo. A multi-server oblivious dynamic searchable encryption framework. *Journal of Computer Security*, 27:1–28, 09 2019.

[43] Qiong Huang and Hongbo Li. An efficient public-key searchable encryption scheme secure against inside keyword guessing attacks. *Information Sciences*, 403-404:1–14, 2017.

[44] Sherman S. M. Chow (The Chinese University of Hong Kong) Jiafan Wang (Data61, CSIRO). Unus pro omnibus: Multi-client searchable encryption via access control. *31st Annual Network and Distributed System Security Symposium, NDSS*, 2024.

[45] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Jamie DeMaria, Andrew Park, and Amos Treiber. Maple: Markov process leakage attacks on encrypted search. Cryptology ePrint Archive, Paper 2023/810, 2023. https://eprint.iacr.org/2023/810.

[46] Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In *International Conference on the Theory and Application of Cryptographic Techniques*, 2019.

[47] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. Structured encryption and leakage suppression. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, pages 339–370, Cham, 2018. Springer International Publishing.

[48] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 965–976, New York, NY, USA, 2012. Association for Computing Machinery.

[49] Aggelos Kiayias, Ozgur Oksuz, Alexander Russell, Qiang Tang, and Bing Wang. Efficient encrypted keyword search for multi-user data sharing. In *European symposium on research in computer security*, pages 173–195. Springer, 2016.

[50] Tung Le, Rouzbeh Behnia, Jorge Guajardo, and Thang Hoang. MUSES: Efficient Multi-user Searchable Encrypted Database. In *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA, August 2024. USENIX Association.

[51] Tung Le and Thang Hoang. MAPLE: A Metadata-Hiding Policy-Controllable Encrypted Search Platform with Minimal Trust. In *Proceedings on Privacy Enhancing Technologies Symposium (PETS)*, pages 184–203, 2023.

[52] Hongbo Li, Qiong Huang, Jianye Huang, and Willy Susilo. Public-key authenticated encryption with keyword search supporting constant trapdoor generation and fast search. *IEEE Transactions on Information Forensics and Security*, 18:396–410, 2023.

[53] Zheli Liu, Yanyu Huang, Xiangfu Song, Bo Li, Jin Li, Yali Yuan, and Changyu Dong. Eurus: Towards an efficient searchable symmetric encryption with size pattern protection. *IEEE Transactions on Dependable and Secure Computing*, 19(3):2023–2037, 2022.

[54] Fucai Luo, Haiyan Wang, Changlu Lin, and Xingfu Yan. Abaeks: Attribute-based authenticated encryption with keyword search over outsourced encrypted data. *IEEE Transactions on Information Forensics and Security*, 18:4970–4983, 2023.

[55] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296. IEEE, 2018.

[56] Priyanka Mondal, Javad Ghareh Chamani, Ioannis Demertzis, and Dimitrios Papadopoulos. I/O-Efficient dynamic searchable encryption meets forward & backward privacy. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2527–2544, Philadelphia, PA, August 2024. USENIX Association.

[57] Mahnaz Noroozi and Ziba Eslami. Public key authenticated encryption with keyword search: revisited. *IET Information Security*, 13(4):336–342, 2019.

[58] Simon Oya and Florian Kerschbaum. IHOP: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2407–2424, Boston, MA, August 2022.

[59] Sikhar Patranabis, Yash Shrivastava, and Debdeep Mukhopadhyay. Provably secure key-aggregate cryptosystems with broadcast aggregate keys for online data sharing on the cloud. *IEEE Transactions on Computers*, 66(5):891–904, 2017.

[60] Baodong Qin, Yu Chen, Qiong Huang, Ximeng Liu, and Dong Zheng. Public-key authenticated encryption with keyword search revisited: Security model and constructions. *Information Sciences*, 516:515–528, 2020.

[61] Zhiwei Shang, Simon Oya, Andreas Peter, and Florian Kerschbaum. Obfuscated access and search patterns in searchable encryption. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021*, 2021.

[62] Daniel Slamanig and Christoph Striecks. Revisiting updatable encryption: Controlled forward security, constructions and a puncturable perspective. In Guy Rothblum and Hoeteck Wee, editors, *Theory of Cryptography*, pages 220–250, Cham, 2023. Springer Nature Switzerland.

[63] Dawn Xiaoding Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 44–55, 2000.

[64] Qiyang Song, Zhuotao Liu, Jiahao Cao, Kun Sun, Qi Li, and Cong Wang. Sap-sse: Protecting search patterns and access patterns in searchable symmetric encryption. *IEEE Transactions on Information Forensics and Security*, 16:1795–1809, 2021.

[65] Xiangfu Song, Yu Zheng, Jianli Bai, Changyu Dong, Zheli Liu, and Ee-Chien Chang. Disco: Dynamic searchable encryption with constant state. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, page 1724–1738, New York, NY, USA, 2024. Association for Computing Machinery.

[66] Shi-Feng Sun, Xingliang Yuan, Joseph K Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 763–780, 2018.

[67] Viet Vo, Shangqi Lai, Xingliang Yuan, Surya Nepal, and Joseph K. Liu. Towards efficient and strong backward private searchable encryption with secure enclaves. In *Applied Cryptography and Network Security: 19th International Conference, ACNS 2021, Kamakura, Japan, June 21–24, 2021, Proceedings, Part I*, page 50–75, Berlin, Heidelberg, 2021. Springer-Verlag.

[68] Viet Vo, Shangqi Lai, Xingliang Yuan, Shi-Feng Sun, Surya Nepal, and Joseph K. Liu. Accelerating forward and backward private searchable encryption using trusted execution. In *Applied Cryptography and Network Security: 18th International Conference, ACNS 2020, Rome, Italy, October 19–22, 2020, Proceedings, Part II*, page 83–103, Berlin, Heidelberg, 2020. Springer-Verlag.

[69] Viet Vo, Xingliang Yuan, Shi-Feng Sun, Joseph K. Liu, Surya Nepal, and Cong Wang. Shielddb: An encrypted document database with padding countermeasures. *IEEE Transactions on Knowledge and Data Engineering*, 35(4):4236–4252, 2023.

[70] Jiafan Wang and Sherman S. M. Chow. Omnes pro uno: Practical Multi-Writer encrypted database. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2371–2388, Boston, MA, August 2022. USENIX Association.

[71] Mingyue Wang, Yinbin Miao, Yu Guo, Hejiao Huang, Cong Wang, and Xiaohua Jia. Aesm2 attribute-based encrypted search for multi-owner and multi-user distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):92–107, 2023.

[72] Lei Xu, Leqian Zheng, Chengzhi Xu, Xingliang Yuan, and Cong Wang. Leakage-abuse attacks against forward and backward private searchable symmetric encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 3003–3017, New York, NY, USA, 2023. Association for Computing Machinery.

[73] Lei Xu, Anxin Zhou, Huayi Duan, Cong Wang, Qian Wang, and Xiaohua Jia. Toward full accounting for leakage exploitation and mitigation in dynamic encrypted databases. *IEEE Transactions on Dependable and Secure Computing*, 21(4):1918–1934, 2024.

[74] Lingling Xu, Wanhua Li, Fangguo Zhang, Rong Cheng, and Shaohua Tang. Authorized keyword searches on public key encrypted data with time controlled keyword privacy. *IEEE Transactions on Information Forensics and Security*, 15:2096–2109, 2020.

[75] Peng Xu, Qianhong Wu, Wei Wang, Willy Susilo, Josep Domingo-Ferrer, and Hai Jin. Generating searchable public-key ciphertexts with hidden structures for fast keyword search. *IEEE Transactions on Information Forensics and Security*, 10(9):1993–2006, 2015.

[76] Ming Zeng, Haifeng Qian, Jie Chen, and Kai Zhang. Forward secure public key encryption with keyword search for outsourced cloud storage. *IEEE Transactions on Cloud Computing*, 10(1):426–438, 2019.

[77] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of File-Injection attacks on searchable encryption. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 707–720, Austin, TX, August 2016.

Appendix A. Security Proofs

A.1. Proof of HICKAE (Theorem 1)

Definition 9 (Bilinear Diffie-Hellman). *Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ be groups of prime order q, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$ be a bilinear pairing. Let P_1 be a generator of $\mathbb{G}_1$ and P_2 be a generator of $\mathbb{G}_2$. The Bilinear Diffie-Hellman problem (BDHP) is the following: Given $aP_1 \in \mathbb{G}_1$ and $b_1P_2, b_2P_2 \in \mathbb{G}_2$ for some $a, b_1, b_2 \in \mathbb{Z}_q$, compute $e(P_1, P_2)^{ab_1b_2}$.*

Lemma 1. HICKAE *is* IND-CPA*-secure by Definition 8 under the BDHP assumption by Definition 9.*

Proof. Given an IND-CPA adversary $\mathcal{A}$, we build $\mathcal{B}$ below that solves the Bilinear Diffie-Hellman problem (BDHP), *i.e.*, computes $[ab_1b_2]_t$ from $([a]_1, [b_1]_1, [b_1]_2, [b_2]_2)$.

1) Sample $\hat{i} \xleftarrow{\$} [n]$, $\tau \xleftarrow{\$} \mathbb{Z}_p$, and $\gamma, \xi \xleftarrow{\$} \mathbb{Z}_q$. Also, initialize an empty map T.
2) Set $[\delta]_2 \leftarrow [b_1]_2$ (i.e., $\delta = b_1$) and $\mathsf{pk} \leftarrow ([\gamma]_2, [\delta]_2, [\xi]_2)$. Sample $\hat{\sigma} \xleftarrow{\$} \mathbb{Z}_p$, then initialize $\mathsf{pp} \leftarrow \{[p_i]_2\}_{i\in[n]}$, where $[p_i]_2 \leftarrow [\alpha^{-\hat{\sigma}^i}]_2$, for $i \in [n]$.
3) Execute HICKAE.IGen for classes $i \in [n]$ to obtain class secrets $\{\sigma'_i\}_{i\in[n]}$, class-binding keys $\{[\mathsf{ek}_i]_2\}_{i\in[n]}$, then execute HICKAE.Prep to obtain the shared secrets and correlation set (Σ, Corr).
4) Simulate the oracle G by lazy programming, *i.e.*, on a new query $[u]_t$, return $G([u]_t) \leftarrow v$, where $v \xleftarrow{\$} \{0,1\}^\lambda$.
5) Upon H query on id:
   a) If $(\mathsf{id}, [h]_1, x, \theta)$ exists in T, return $[h]_1$.
   b) Otherwise, $(\mathsf{id}, [h]_1, x, \theta)$ has not appeared in T, pick $x \xleftarrow{\$} \mathbb{Z}_q$, flip a coin that outputs a bit $\theta = 1$ with probability ρ, $\theta = 0$ otherwise. If $\theta = 1$, return $[h]_1 \leftarrow [x]_1$. Otherwise, $\theta = 0$, return $[h]_1 \leftarrow x[a]_1$. Record $(\mathsf{id}, [h]_1, x, \theta)$ in the map T.
6) Upon $\mathsf{Enc}\mathcal{O}$ query on plaintext m, class identifier i and embedded identity id as input:
   a) Retrieve $(\mathsf{id}, [h]_1, x, \theta)$ from the map T.
   b) Sample $r \xleftarrow{\$} \mathbb{Z}_q$, set $[c_1]_2 \leftarrow [r]_2$, $[c_2]_2 \leftarrow r[\xi]_2$, $[c_3]_2 \leftarrow r([\gamma]_2 + [\mathsf{ek}_i]_2)$, and $c_4 \leftarrow m \oplus G(r[h]_1[\delta]_2)$.
   c) Return $c \leftarrow ([c_1]_2, [c_2]_2, [c_3]_2, c_4)$.
7) Upon $\mathsf{Ext}\mathcal{O}$ query on $S \subseteq [n]$ and id:
   a) If $\hat{i} \notin S$, get $(\mathsf{id}, [h]_1, x, \theta)$ from T, return $(k_1, [k_2]_1, [k_3]_1)$, where $k_1 \leftarrow \alpha^{\tau'} + \xi$, with $\tau' \xleftarrow{\$} \mathbb{Z}_p$,

$[k_2]_1 \leftarrow \gamma \sum_{j\in S}[\alpha^{\tau+\sigma_j}]_1 + [\alpha^{\tau}]_1 + x[b_1]_1\alpha^{-\tau'}$, and $[k_3]_1 \leftarrow \sum_{j\in S}[\alpha^{\tau+\tau'+\sigma_j}]_1$.

b) Otherwise $\hat{i} \in S$, retrieve $(\mathsf{id}, [h]_1, x, \theta)$ from the map T. If $\theta = 1$, return $(k_1, [k_2]_1, [k_3]_1)$, where $k_1 \leftarrow \alpha^{\tau'} + \xi$, with $\tau' \xleftarrow{\$} \mathbb{Z}_p$, $[k_2]_1 \leftarrow \gamma \sum_{j\in S}[\alpha^{\tau+\sigma_j}]_1 + [\alpha^{\tau}]_1 + x[b_1]_1\alpha^{-\tau'}$, and $[k_3]_1 \leftarrow \sum_{j\in S}[\alpha^{\tau+\tau'+\sigma_j}]_1$. Otherwise, abort and output a random $\mathbb{G}_t$ element.

8) Receive $(\mathsf{st}, m_0, m_1, i^*, \mathsf{id}^*) \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{pk})$.
9) If $\hat{i} \neq i^*$, abort and output a random $\mathbb{G}_t$ element.
10) First query for $H(\mathsf{id}^*)$ to obtain $(\mathsf{id}^*, [h^*]_1, x^*, \theta)$ from T.
11) If $\theta = 1$, abort.
12) Simulate the ciphertext as follows:
    a) Sample $b \xleftarrow{\$} \{0,1\}$. Set $[c_1]_2 \leftarrow [b_2]_2$, $[c_2]_2 \leftarrow [b_2]_2\xi$, and $[c_3]_2 \leftarrow [b_2]_2(\gamma + \alpha^{-\sigma_{i^*}})$.
    b) Set $c_4 \leftarrow m_b \oplus v_b^*$ with $v_b^* \xleftarrow{\$} \{0,1\}^\lambda \ \forall\ b \in \{0,1\}$.
    c) Return $c^* \leftarrow ([c_1]_2, [c_2]_2, [c_3]_2, c_4)$.
13) Receive $b' \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{st}, c^*)$.
14) Randomly pick one query $[u^*]_t$ to the G oracle.
15) Output $[u^*]_t/x^*$.

If $\mathcal{B}$ does not abort, it simulates the IND-CPA experiment for $\mathcal{A}$ successfully. For $\mathcal{A}$ to win the game, $\mathcal{A}$ must have queried G on $b_2[h^*]_1[\delta]_2$, which matches how encryption is done, for otherwise b is information-theoretically hidden from $\mathcal{A}$. $\mathcal{B}$ can therefore extract $[h^* b_2 \delta]_t/x^* = [ab_1b_2]_t$ as a BDHP solution, with probability $1/q_G$, where q_G is the number of oracle queries to G that $\mathcal{A}$ made.

It remains to analyze the probability that algorithm $\mathcal{B}$ does not abort during the simulation. Suppose $\mathcal{A}$ makes a total of q_E $\mathsf{Ext}\mathcal{O}$ queries with $\hat{i} \in S$, then the probability that $\mathcal{B}$ does not abort is ρ^{q_E}. The probability that it does not abort during the challenge step is $(1-\rho)/n$. Therefore, the probability that $\mathcal{B}$ does not abort during the simulation is $\rho^{q_E}(1-\rho)/n$. This value is maximized at $\rho_{\mathsf{opt}} = 1 - 1/(q_E+1)$. Using ρ_{opt}, the probability that $\mathcal{B}$ does not abort is at least $1/ne(1+q_E)$. □

Lemma 2. HICKAE *is* IND-ANON*-secure by Definition 8 under the BDHP assumption by Definition 9.*

Proof. Given an IND-ANON adversary $\mathcal{A}$, we build $\mathcal{B}$ below that solves the Bilinear Diffie-Hellman problem (BDHP), *i.e.*, computes $[ab_1b_2]_t$ from $([a]_1, [b_1]_1, [b_1]_2, [b_2]_2)$.
Repeat steps 1-7 in the above proof for Lemma 1.
8) Receive $(\mathsf{st}, m^*, i^*, \mathsf{id}_0^*, \mathsf{id}_1^*) \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{pk})$.
9) If $\hat{i} \neq i^*$, abort and output a random $\mathbb{G}_t$ element.
10) First query for $H(\mathsf{id}_0^*)$ and $H(\mathsf{id}_1^*)$ to obtain $(\mathsf{id}_0^*, [h_0^*]_1, x_0^*, \theta_0)$ and $(\mathsf{id}_1^*, [h_1^*]_1, x_1^*, \theta_1)$ from T.
11) If $\theta_{b^*} = 1$ for any $b^* \in \{0,1\}$, abort.
12) Simulate the ciphertext as follows:
    a) Sample $b \xleftarrow{\$} \{0,1\}$. Set $[c_1]_2 \leftarrow [b_2]_2$, $[c_2]_2 \leftarrow [b_2]_2\xi$, and $[c_3]_2 \leftarrow [b_2]_2(\gamma + \alpha^{-\sigma_{i^*}})$.
    b) Set $c_4 \leftarrow m^* \oplus v_b^*$ with $v_b^* \xleftarrow{\$} \{0,1\}^\lambda \ \forall\ b \in \{0,1\}$.
    c) Return $c^* \leftarrow ([c_1]_2, [c_2]_2, [c_3]_2, c_4)$.
13) Receive $b' \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{st}, c^*)$.
14) Randomly pick one query $[u^*]_t$ to the G oracle.
15) Output $[u^*]_t/x_{b'}^*$.

If $\mathcal{B}$ does not abort, it simulates the IND-ANON experiment for $\mathcal{A}$ successfully. For $\mathcal{A}$ to win the game, $\mathcal{A}$ must have queried G on $b_2[h_{b'}^*]_1[\delta]_2$, which matches how encryption is done, for otherwise b is information-theoretically hidden from $\mathcal{A}$. $\mathcal{B}$ can therefore extract $[h_{b'}^* b_2 \delta]_t/x_{b'}^* = [ab_1b_2]_t$ as a BDHP solution, with probability $1/q_G$, where q_G is the number of oracle queries to G that $\mathcal{A}$ made.

Suppose $\mathcal{A}$ makes a total of q_E $\mathsf{Ext}\mathcal{O}$ queries with $\hat{i} \in S$, then the probability that $\mathcal{B}$ does not abort is ρ^{q_E}. The probability that it does not abort during the challenge step is $(1-\rho)^2/n$. Therefore, the probability that $\mathcal{B}$ does not abort during the simulation is $\rho^{q_E}(1-\rho)^2/n$. □

Definition 10 (Discrete-Log). *Let $\mathbb{G}$ be a group of prime order q, and P be a generator of $\mathbb{G}$. The Discrete-Log problem (DLP) is the following: Given $Q \in \mathbb{G}$, compute the smallest positive integer $a \in \mathbb{Z}_q$ such that $aP = Q$.*

Lemma 3. HICKAE *is* IND-CIA*-secure by Definition 8 under the DLP assumption by Definition 10.*

Proof. Given an IND-CIA adversary $\mathcal{A}$, we build $\mathcal{B}$ below that solves the Discrete-Log problem (DLP), *i.e.*, computes $a \in \mathbb{Z}_q$ from $[a]_2$ and $[1]_2$.

1) Sample $\hat{i} \xleftarrow{\$} [n]$, $\tau \xleftarrow{\$} \mathbb{Z}_p$, and $\gamma, \delta, \xi \xleftarrow{\$} \mathbb{Z}_q$. Also, initialize an empty map T.
2) Set $\mathsf{pk} \leftarrow ([\gamma]_2, [\delta]_2, [\xi]_2)$. Sample $\hat{\sigma} \xleftarrow{\$} \mathbb{Z}_p$, then initialize $\mathsf{pp} \leftarrow \{[p_i]_2\}_{i\in[n]}$, where $[p_i]_2 \leftarrow [\alpha^{-\hat{\sigma}^i}]_2$, for $i \in [n]$.
3) Execute HICKAE.IGen for classes $i \in [n]$ to obtain class secrets $\{\sigma_i'\}_{i\in[n]}$, class-binding keys $\{[\mathsf{ek}_i]_2\}_{i\in[n]}$, then execute HICKAE.Prep to obtain the shared secrets and correlation set (Σ, Corr).
4) Simulate the oracle G by lazy programming, *i.e.*, on a new query $[u]_t$, return $G([u]_t) \leftarrow v$, where $v \xleftarrow{\$} \{0,1\}^\lambda$.
5) Upon H query on id:
    a) If $(\mathsf{id}, [h]_1, \theta)$ exists in T, return $[h]_1$.
    b) Otherwise, $(\mathsf{id}, [h]_1, \theta)$ has not appeared in T, pick $x \xleftarrow{\$} \mathbb{Z}_q$, flip a coin that outputs a bit $\theta = 1$ with probability ρ, $\theta = 0$ otherwise. If $\theta = 1$, return $[h]_1 \leftarrow x[\delta]_1$. Otherwise, $\theta = 0$, return $[h]_1 \leftarrow [x]_1$. Record $(\mathsf{id}, [h]_1, \theta)$ in the map T.
6) Upon $\mathsf{Enc}\mathcal{O}$ query on plaintext m, class identifier i and embedded identity id as input:
    a) Retrieve $(\mathsf{id}, [h]_1, \theta)$ from the map T.
    b) Sample $r \xleftarrow{\$} \mathbb{Z}_q$, set $[c_1]_2 \leftarrow [r]_2$, $[c_2]_2 \leftarrow r[\xi]_2$, $[c_3]_2 \leftarrow r([\gamma]_2 + [\mathsf{ek}_i]_2)$, and $c_4 \leftarrow m \oplus G(r[h]_1[\delta]_2)$.
    c) Return $c \leftarrow ([c_1]_2, [c_2]_2, [c_3]_2, c_4)$.
7) Upon $\mathsf{Ext}\mathcal{O}$ query on $S \subseteq [n]$ and id:
    a) Retrieve $(\mathsf{id}, [h]_1, \theta)$ from the map T.
    b) Return $(k_1, [k_2]_1, [k_3]_1)$, where $k_1 \leftarrow \alpha^{\tau'} + \xi$, with $\tau' \xleftarrow{\$} \mathbb{Z}_p$, $[k_2]_1 \leftarrow \gamma \sum_{j\in S}[\alpha^{\tau+\sigma_j}]_1 + [\alpha^{\tau}]_1 + [h]_1\alpha^{-\tau'}$, and $[k_3]_1 \leftarrow \sum_{j\in S}[\alpha^{\tau+\tau'+\sigma_j}]_1$.
8) Receive $(\mathsf{st}, i^*, \mathsf{id}_0^*, \mathsf{id}_1^*) \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{pk})$
9) If $\hat{i} \neq i^*$, abort and output a random $\mathbb{Z}_q$ element.

10) First query for $H(\mathsf{id}_0^*)$ and $H(\mathsf{id}_1^*)$ to obtain $(\mathsf{id}_0^*, [h_0^*]_1, \theta_0)$ and $(\mathsf{id}_1^*, [h_1^*]_1, \theta_1)$ from T.
11) If $\theta_{b^*} = 1$ for any $b^* \in \{0,1\}$, abort.
12) Simulate the aggregated key as follows:
    a) Sample $b \xleftarrow{\$} \{0,1\}$.
    b) Set $k_1 \leftarrow \alpha^{\tau'} + \xi$, with $\tau' \xleftarrow{\$} \mathbb{Z}_p$, $[k_2]_1 \leftarrow \gamma[\alpha^{\tau+\sigma_{i^*}}]_1 + [\alpha^{\tau}]_1 + [h_b]_1\alpha^{-\tau'}$, and $[k_3]_1 \leftarrow [\alpha^{\tau+\tau'+\sigma_{i^*}}]_1$.
    c) Return $\mathsf{ak}^* \leftarrow (k_1, [k_2]_1, [k_3]_1)$.
13) Receive $b' \leftarrow \mathcal{A}^{\mathsf{Ext}\mathcal{O},\mathsf{Enc}\mathcal{O}}(\mathsf{st}, \mathsf{ak}^*)$.

If $\mathcal{B}$ does not abort, it simulates the IND-CIA experiment for $\mathcal{A}$ successfully. For $\mathcal{A}$ to win the game, $\mathcal{A}$ must find $[\alpha^{\tau}]_1$ to evaluate $([k_2]_1 - [\alpha^{\tau}]_1)(k_1[1]_2 - [\xi]_2) - [\gamma]_2[k_3]_1 = [h_{b'}^*]_1[\delta]_2$, and then compare it with $[h_0^*]_1[\delta]_2$ and $[h_1^*]_1[\delta]_2$ to know whether id_0^* or id_1^* is chosen as the challenge. Otherwise b is information-theoretically hidden from $\mathcal{A}$. With any two corrupt classes j_1 and j_2, their identity correlation values are $[\alpha^{\tau+\sigma_{j_1}-\sigma_{j_2}}]_1$, $[\alpha^{\tau+\sigma_{j_2}-\sigma_{j_1}}]_1$, and their identity-binding keys are $[\alpha^{-\sigma_{j_1}}]_2$, $[\alpha^{-\sigma_{j_2}}]_2$, then $[\alpha^{\tau}]_1$ can be obtained if $\alpha^{-\sigma_{j_1}}$ and $\alpha^{-\sigma_{j_1}}$ are revealed, which happen if and only if the discrete-log problem were solved.

Similar to the analysis in the proof of Lemma 2, the probability that $\mathcal{B}$ does not abort is also $\rho^{q_E}(1-\rho)^2/n$, where q_E is the total number of $\mathsf{Enc}\mathcal{O}$ queries that $\mathcal{B}$ made. □

Putting it together. Lemma 1, Lemma 2 and Lemma 3 prove the IND-CPA, IND-ANON and IND-CIA security of HICKAE, respectively. Together these complete the proof of Theorem 1.

A.2. Proof of Hermes (Theorem 2)

Proof. Let $\mathcal{L}_{\mathsf{sse},i} = \{\mathcal{L}^{\mathsf{Setup}}_{\mathsf{sse},i}, \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}, \mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}\}$ and $\mathcal{S}_{\mathsf{sse},i}$ be the leakage functions and the simulator of the i-th DSSE instance for $i \in [n]$, respectively. The initially empty EIDX_i and ESTkn_i leak nothing. $\mathcal{L}^{\mathsf{Setup}}_{\mathsf{hse}}$ only leaks the number of writers and their identifiers.

Let $W_{\mathsf{Srch}}(i,e) = \{w : (\mathsf{Srch}, w, S, e) \in \mathcal{H} \wedge i \in S\}$ be the set of keywords that has been searched at epoch e for writer subsets containing i. We parse DSSE update leakage as $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}(\mathsf{op}, w, f) = \{\varnothing\}$, if $w \notin W_{\mathsf{Srch}}(i,e)$; and $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}(\mathsf{op}, w, f) = \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w)$, otherwise. Let H': $\{0,1\}^* \to [0, \sqrt{W} - 1]$ be a public keyword-partition mapping.

We derive a $(q+1)$-hybrid sequence starting from $\mathbf{Hybrid}_0 = \mathsf{IND}^0_{\mathsf{MSE}}$, and the last hybrid $\mathbf{Hybrid}_q$ is exactly $\mathsf{IND}^1_{\mathsf{MSE}}$. For $\ell \in \{0, \ldots, q-1\}$, the only difference between $\mathbf{Hybrid}_\ell$ and $\mathbf{Hybrid}_{\ell+1}$ is that the oracle responds to the $(\ell+1)$-th query in $\mathbf{Hybrid}_\ell$ with input $b = 0$, while responding to the $(\ell+1)$-th query in $\mathbf{Hybrid}_{\ell+1}$ with input $b = 1$. The oracles implicitly take the current epoch e as an input.

We prove that $\mathcal{A}$ cannot distinguish $\mathsf{IND}^0_{\mathsf{MSE}}$ from $\mathsf{IND}^1_{\mathsf{MSE}}$ with non-negligible probability by showing that each hybrid (except the first) is indistinguishable from its previous.

For $\ell \in \{0, \ldots, q-1\}$, $\mathsf{Hist}_{\ell+1}$ can fall into three cases:

(*1*) $\mathsf{Crpt}\mathcal{O}_b$ on (i_0, i_1): It will only be answered when writer identifier $i = i_0 = i_1$. Since the information related to the corrupt writer is revealed, it requires that the tuples updated by writer i (*i.e.*, $\mathsf{UpdtBy}(\mathsf{i})$) are the same for either $b = 0$ or $b = 1$. We have $\mathbf{Hybrid}_\ell = \mathbf{Hybrid}_{\ell+1}$ as the views of $\mathcal{A}$ are identical in this case.

(*2*) $\mathsf{Srch}\mathcal{O}_b$ on $(\{S_k, w_k\}_{k\in\{0,1\}})$: As the target writers of any search leaks, it will only be answered when $S = S_0 = S_1$. If there exists some writer $i \in S$ such that $(\mathsf{Crpt}\mathcal{O}, i) \in \mathcal{H}$, the keyword to be searched is revealed, the oracle only answers the query when w_0 and w_1 are identical. In this case, the oracle simply invokes the simulator $\mathcal{S}_{\mathsf{sse},i}$ with $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w)$ to simulate DSSE search regarding the DSSE trapdoor of w encrypted with HICKAE during the previous $\mathsf{Updt}\mathcal{O}$. The challenger returns the HICKAE keyword decryption key of $(S, w\|\mathbf{t})$ (detailed in §5.2.2) as $\mathfrak{s}_w$, and the HICKAE partition decryption key of $(S, H'(w))$ as $\mathfrak{s}_p$. With recursive partitioning, the challenger executes similarly but for more partition levels. The indistinguishability between $\mathbf{Hybrid}_\ell$ and $\mathbf{Hybrid}_{\ell+1}$ is guaranteed by IND-CIA security of HICKAE and $\mathcal{L}_{\mathsf{sse}}$-adaptive security of DSSE.

(*3*) $\mathsf{Updt}\mathcal{O}_b$ on $(\{i_k, \mathsf{op}_k, w_k, f_k\}_{k\in\{0,1\}})$: The oracle answers the queries when $i = i_0 = i_1$, as the writer identifier will be leaked during update. Obviously, if writer i has been corrupt, $\mathcal{A}$ will have the knowledge of the update tuples, and the oracle will only be answered when two update tuples are identical in this case.

Depending on whether or not both keywords w_0 and w_1 have been searched over class i at current epoch e, there are two cases as follows:

- If $w_0 \in W_{\mathsf{Srch}}(i,e) \wedge w_1 \in W_{\mathsf{Srch}}(i,e)$, it is required that their update leakages are identical, which is essentially $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}$ as $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i} = \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}$ for this case. It typically requires $w = w_0 = w_1$. The oracle invokes DSSE simulator $\mathcal{S}_{\mathsf{sse}}$ with $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w)$ to simulate $\mathfrak{u}_{\mathsf{sse}}$.
- Otherwise, the oracle invokes the DSSE simulator $\mathcal{S}_{\mathsf{sse}}$ with $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}_b, w_b, f_b) = \{\varnothing\}$ to simulate $\mathfrak{u}_{\mathsf{sse}}$.

The oracle calls the DSSE simulator $\mathcal{S}_{\mathsf{sse},i}$ to simulate i's DSSE keyword trapdoor $\mathfrak{s}_{\mathsf{sse}}$, which is HICKAE-encrypted as c_w, treating $w_b\|\mathbf{t}$, where $\mathbf{t} \in \Gamma_{\mathbf{t}(e)}$ (see §5.2.2), as the embedded identity. Also, the oracle takes $\mathsf{paddr} \leftarrow \mathsf{T}_{\mathsf{paddr}}[H'(w)]$ if $\mathsf{T}_{\mathsf{paddr}}[H'(w)] \neq 0^\lambda$. Otherwise, it picks a μ-bit string as paddr and puts it in $\mathsf{T}_{\mathsf{paddr}}[H'(w)]$, then outputs an encrypted partition token c_p for paddr, with $H'(w)$ as the embedded identity.

The indistinguishability between $\mathbf{Hybrid}_\ell$ and $\mathbf{Hybrid}_{\ell+1}$ is guaranteed by IND-CPA and IND-ANON security of HICKAE, and $\mathcal{L}_{\mathsf{sse}}$-adaptive security of DSSE.

By repeating the above procedure for $\ell \in [q-1]$, we conclude that $\mathcal{A}$ cannot distinguish $\mathbf{Hybrid}_0 = \mathsf{IND}^0_{\mathsf{MSE}}$ from $\mathbf{Hybrid}_q = \mathsf{IND}^1_{\mathsf{MSE}}$. Thus Hermes is $\mathcal{L}_{\mathsf{mse}}$-adaptively-secure.

Forward privacy of MSE (Definition 6) constrains the update leakage when updating keywords that have not been searched at the same epoch. In Hermes, for this case (i.e., $w \notin \mathsf{WSrch}(i,e)$), the leakage $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{mse}}$ is $\{i\}$, fulfilling Definition 6. Thus, Hermes is forward-private. □