Omnes pro uno: Practical Multi-Writer Encrypted Database

Jiafan Wang
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Sherman S. M. Chow*
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Abstract

Multi-writer encrypted databases allow a reader to search over data contributed by multiple writers securely. Public-key searchable encryption (PKSE) appears to be the right primitive. However, its search latency is not welcomed in practice for requiring public-key operations linear in the database size. In contrast, symmetric searchable encryption (SSE) realizes sublinear search, but it is inherently not multi-writer.

This paper aims for the best of both SSE and PKSE, *i.e.*, sublinear search and multiple writers, by formalizing hybrid searchable encryption (HSE), with some seemingly conflicting yet desirable features, requiring new insights to achieve. HSE, built on top of dynamic SSE (DSSE), should satisfy the *de facto* standard of forward privacy. Its multi-writer support makes the known approach (of secret state maintenance) fail. HSE should also feature confined search, ideally with search tokens of size independent of the writer subset size for each search. For these, we devise a partial rebuild technique and two building blocks (of independent interests) – identity-coupling key-aggregate encryption and epoch-based forward-private DSSE. Our evaluation over real-world datasets shows that HSE surpasses prior arts by orders of magnitude.

1 Introduction

Searchable encryption enables a "reader" to search over databases encrypted by "writers" and stored in a remote server. To search means identifying all files containing a keyword w by granting a search token for w to the server. Since the seminal work by Song *et al.* [34], many schemes appear, which broadly fall into two categories: symmetric searchable encryption (SSE) and public-key searchable encryption (PKSE) [7].

SSE mainly considers a client takes both the reader and writer roles, while the server is honest-but-curious. The client creates an encrypted database and later searches over it using a secret. With pre-built indices, the search complexity of typical SSE schemes is *sublinear in the database size* or optimally linear in the number of matches [19]. Minimizing client-server interactions (to one round) is often a major goal (*e.g.* [33]).

Kamara *et al.* [26] proposed dynamic SSE (DSSE), where a client could update the outsourced database by creating an update token. The new operation introduces new attack surfaces: it is deemed insecure if any search token previously delegated for a certain keyword can identify any newly inserted file, leading to injection attacks [41]. *Forward privacy* thus becomes a *de facto* standard [10, 29, 35]. It requires the past search not to compromise the privacy of future data. Forward-private DSSE typically requires the client to *refresh* the secret state (which could be outsourced at the cost of interactive update [20] or one more round trip before each search). Otherwise, search tokens and update tokens remain the same for all time, no matter whether a search has happened or not.

PKSE is for *multiple writers*, taking a reader public key to generate searchable ciphertexts. A search token from the reader allows the server to search over these ciphertexts. In its motivating application [7], a reader lets the gateway search over encrypted emails from writers, *e.g.*, those labeled with "urgent." One can obtain PKSE from anonymous identity-based encryption (IBE), viewing keywords as identities [2].

Techniques for sublinear search time in DSSE seem inapplicable to PKSE. The lack of shared secrets between writers forbids them from jointly constructing a search index, which is essential for sublinear search. Thus, the search time of most PKSE schemes is undesirably linear in the database size. Unfortunately, there is hardly any effort in improving the searching time needed for real-world applications in these two decades [34], probably due to the inherent complexity. With very different features and focuses of SSE and PKSE, a challenging and practically relevant question is thus raised:

Can we achieve the best of both worlds – sublinear search in symmetric searchable encryption and multi-writer support in public-key searchable encryption – while featuring forward privacy and non-interactive searches and updates?

*Corresponding author: Supported by GRF (CUHK 14210217) of UGC. We are grateful to Seny Kamara for encouragement during the infancy stage when the hybrid idea is first conceived before the birth of DSSE, to Russell W. F. Lai for his help during the early stages, and to the anonymous reviewers.

1.1 Multi-Writer Encrypted Database via HSE

Our affirmative answer is a new notion we call *hybrid searchable encryption* (HSE). While server-side efficiency is our motivating concern, there can be different deployment expectations. We consider several motivating canonical use-cases of multi-writer applications to illustrate the desideratum of HSE: sensor networks [31], medical databases [18], and contributive applications (*e.g.*, email [7] and machine learning [28]).

No Secret-Key Distribution. Sharing a symmetric key with each writer is a bit (managerially/monetarily) costly, especially for weak devices. Requiring writing tokens simply does not fit with emails and contributive applications, discouraging "unmet writers" from contributing. HSE writers should *only use the public key* of the reader to create HSE ciphertexts.

No Synchronous Communication. Communication rounds, an important criterion of searchable encryption [19], should be kept minimal. Connectivity is relatively scarce in sensor networks and mobile IoT networks, and round-trip latency can be large. A usable scheme would not rely on prompt responses of the reader to each writer before they can contribute data.

Optimal Asymmetric and Symmetric Efficiency. HSE writers can *independently* (public-key) encrypt their updates for the *exact or different* keywords. Thus, we *must* use a number of public-key operations to differentiate them. Realistically, it should be linear in the number of *active keywords* added by the writers that the reader will search over, which is typically less than the number of updates. Meanwhile, HSE still keeps the number of *symmetric operations* sublinear as optimal SSE.

Compact Token for Confined Search. HSE can restrict the scope of each search to a different writer subset. This prevents unnecessary leakage and accelerates the search. To save computation and bandwidth non-trivially, the search token size should be *independent* of the number of interested writers.

Forward Privacy. The impact of not having forward privacy is amplified in HSE – the search authorized over an old database contributed by some writers should not automatically compromise the data privacy of the future data submitted by other honest writers. The notion is well-studied in DSSE as a property centered around a *single* client, who can, after every search, *immediately* refresh a secret (local) state. Nevertheless, in HSE, writers do not know what the reader has done unless they synchronize with the reader or the server before every update. Trivial adoption of known techniques means sharing the secret with all writers, nullifying forward privacy against writers. It is now apparent that the requirements of HSE are inter-related, *e.g.*, non-interactiveness renders traditional forward privacy tricks no longer applicable. Formulating and achieving it in non-interactive HSE thus appear to be solving the unsolvable and need new insights.

A Warm-Up Construction. We first propose a conceptually simple generic HSE construction G-HSE from DSSE and anonymous IBE as a baseline. The reader initializes n instances of anonymous IBE for n writers, while each writer independently builds a DSSE instance. To avoid name-clashing with the identity of IBE, we associate each *writer* to a *class*.

To update, besides the DSSE update, the writer IBE-encrypts the DSSE search token of the associated keyword w, treating w as the identity, if the token has not been encrypted before. The server stores a set of encrypted DSSE search tokens. To search, the reader generates decryption keys for the keyword to be searched under IBE instances of the target subset of classes. The server decrypts entries in the token set for DSSE search tokens and searches over their DSSE instances.

G-HSE is non-interactive and enjoys sublinear search time of DSSE with an arguably optimal number of public-key operations for searching the keyword set of each target writer.

1.2 Overview of ICKAE, E-DSSE, and FP-HSE

G-HSE fails to provide 1) forward privacy as the search tokens never change; and 2) compact token for confined search as the number of search tokens is linear in the writer-subset size. We thus design FP-HSE, a forward-private HSE construction with constant-size search tokens. It is instantiated with two building blocks, each resolving one kind of shortcomings.

ICKAE. The first one is identity-coupling key-aggregate encryption (ICKAE). As key-aggregate encryption [18], the distinctive advantage of ICKAE is its $O(1)$-size decryption key aggregating the decryption power for any polynomial number of (writer) classes. In the context of FP-HSE, an $O(1)$-size decryption key constrained to *any* subset of writers (totaling $(2^n - 1)$ possibilities for n writers) can be derived.

The ciphertext and the decryption key of ICKAE are coupled with an identity (ID), such that the decryption key for a writer subset and a specific ID can only decrypt the ciphertext created for any writer in the subset and the same ID. The ID associated with the ciphertext is hidden from anyone who cannot decrypt. ICKAE thus enables $O(1)$-size tokens for FP-HSE search. Our FP-HSE uses both the keyword and the "epoch" as an ID, so a decryption key cannot decrypt ciphertexts for subsequent epochs. ICKAE is also of independent interest for its more fine-grained access control than its basis.

E-DSSE. Our second building block is E-DSSE, which adapts the epoch-based forward privacy of public-key encryption (PKE) [12] to epoch-based forward privacy in DSSE. It relies on a loosely synchronized clock indicating the current epoch.

E-DSSE enables the data owner to delegate the *temporal search ability* to other readers or the server via a *constant-size* token, which can retrieve any related tuples updated at the epoch associated with the token and any prior epochs. The search token automatically fails to retrieve updates at any later epochs, without the need for refreshment per search after new updates in forward-private DSSE [10, 29]. This property helps to realize forward privacy in HSE without synchronous communication and comes in handy for multi-reader DSSE.

Scheme	Search Complexity	Token Size	Forward Privacy	Confined Search
PKSE [2]	$O(\lvert\mathsf{DB}\rvert^{\dagger})$	$O(1)$	✗	✗
FP-PKSE [40]	$O(\lvert\mathsf{DB}\rvert^{\dagger})$	$O(1)$	✓	✗
SPCHS [39]	$O(\lvert R_w\rvert^{\dagger})$	$O(1)$	✗	✗
G-HSE	$O(\lvert W\rvert^{\dagger}+\lvert R_w\rvert)$	$O(\lvert S\rvert)$	✗	✓
FP-HSE	$O(\lvert W\rvert^{\dagger}+\lvert R_w\rvert)$	$O(1)$	✓	✓

S: a subset of writers; DB: the whole database; †: public-key operation; W: the set of active keywords written by S; R_w: updates of w from S.

Table 1: Searchable Encryption for Multiple Writers

The epoch length in E-DSSE can be set autonomously. The system thus provides a tunable tradeoff between flexibility and forward privacy, which is *first-of-its-kind* in the SSE literature to our knowledge. A shorter epoch trades flexibility for higher forward privacy as issued tokens become invalid sooner. This also matches epoch-based secure messaging systems [22].

Partial Rebuild. Merely instantiating HSE by ICKAE and E-DSSE is still lacking. Note that the search token changes when updating the same keyword at different epochs in E-DSSE. The number of ciphertexts for search tokens thus needs to be ever-growing to ensure the reader could always get the latest results. This will severely degrade the search performance of FP-HSE. Our remedy is to *partially* rebuild the token parts instead of the whole database by adapting an idea originally proposed for breach resistance [3].

Note that writers still need not be synchronized. They could go offline for a few epochs without harming security.

FP-HSE. FP-HSE precludes the shortcomings of G-HSE: 1) FP-HSE inherits epoch-based forward privacy of E-DSSE as the token changes per epoch; and 2) FP-HSE realizes compact token for confined search due to $O(1)$-size ICKAE decryption keys. Table 1 summarizes HSE's performance and properties.

1.3 Related Work

For easy indexing, early attempts study public-key deterministic encryption [5] with unavoidable weakened security. A notable alternative of "searchable public-key ciphertexts with hidden structures" (SPCHS) was proposed by Xu *et al.* [39]. It can be treated as deriving the secret key for PKSE via non-interactive key exchange. Consequently, each traversal during a search takes pairing operations. More importantly, SPCHS supports neither forward privacy nor confined search.

Many *"multi-user"* searchable encryption schemes have appeared. Their user often refers to a reader. They share SSE search tokens to multiple readers, *e.g.*, via broadcast encryption (BE) [19], proxy re-encryption [24], or relying on trusted hardware [30]. Many key-aggregate searchable encryption schemes have also appeared. They still take linear time, and some are broken. We omit them due to the page limit.

Our ICKAE scheme got inspiration from hierarchical ID-coupling broadcast encryption [4] and KAE [18]. The former notion is extensible to forward-secure BE and searchable BE. In principle, it could lead to forward-secure searchable BE, which predates FP-PKSE [40]. Designing an ID-coupling scheme is non-trivial, *e.g.*, trivial extension from an anonymous IBE scheme may lose its anonymity [4].

1.4 Our Contribution

New Paradigm of Searchable Encryption (Section 3). We propose HSE, a new notion that can simultaneously achieve sublinear search complexity of SSE and multi-writer support in PKSE. It is non-trivial to define its security when there is only one client in DSSE, and writer corruption (meaningful in HSE) is not a concern in PKSE since writers are stateless. Notably, we extend the history-based leakage of DSSE [19] to precisely capture the leakage in the context of HSE, while the security of PKSE often does not involve leakage at all.

New Primitive: ID-Coupling Key-Aggregate Encryption (Section 4). We propose the notion of ICKAE with a scheme featuring $O(1)$-size decryption keys that couples with an ID. Generating an aggregate key for n classes is more efficient than generating n decryption keys for n IBE instances [8].

New Epoch-Based Forward Privacy (Sections 3.4 and 5). We formally define epoch-based forward privacy of HSE and DSSE for a general notion of epoch (*e.g.*, key rotation leads to a new epoch). We propose an epoch-based forward-private DSSE construction E-DSSE with optimal complexities.

Forward-Private HSE with Compact Tokens (Section 6). Our FP-HSE instantiated from new building blocks (ICKAE and E-DSSE) features non-interactive forward privacy and $O(1)$-size tokens for confined searches. It is modular and benefits from better building blocks in the future. Its token-based design is compatible with many existing SSE works. In particular, it can be easily extended with backward privacy.

Experimental Evaluation (Section 7). We implement both G-HSE and FP-HSE, compare their performance with prior arts over real-world datasets for our use-cases, and comprehensively analyze their efficiency under various parameters.

2 Dynamic Symmetric Searchable Encryption

$||$ is concatenation. $x \leftarrow\!\$\, X$ samples x uniformly from set X. $\mathsf{negl}(\lambda)$ denotes a negligible function in λ. $[n] = \{1,\ldots,n\}$.

Definition 1 (DSSE **[26]**) *A* DSSE *scheme consists of five probabilistic polynomial-time (*PPT*) algorithms:*

$(\mathsf{k},\mathsf{st},\mathsf{EDB}) \leftarrow \mathsf{Setup}(1^{\lambda})$ *takes as input a security parameter* λ*. It outputs a secret key* k *and a state* st *to be stored locally and an initialized encrypted database* EDB*.*

$(\mathfrak{s},\mathsf{st}') \leftarrow \mathsf{SrchTkn}(\mathsf{st},\mathsf{k},w)$ *takes a state* st*, a secret key* k*, and a keyword* w*. It outputs a search token* $\mathfrak{s}$ *and a state* st′*.*

$(R,\mathsf{EDB}') \leftarrow \mathsf{Srch}(\mathfrak{s},\mathsf{EDB})$ *takes as input a search token* $\mathfrak{s}$ *and an encrypted database* EDB*. It outputs a possibly updated encrypted database* EDB′ *and the search result* R*.*

$(\mathfrak{u}, \mathsf{st}') \leftarrow \mathsf{UpdtTkn}(\mathsf{st}, \mathsf{k}, \mathsf{op}, w, f)$ *takes as input a state* st*, a secret key* k*, an operation* $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$*, and a keyword-file pair* (w, f)*. It outputs an update token* $\mathfrak{u}$ *and a new state* st'*.*

$\mathsf{EDB}' \leftarrow \mathsf{Updt}(\mathfrak{u}, \mathsf{EDB})$ *takes as input an update token* $\mathfrak{u}$ *and* EDB*. It outputs an updated encrypted database* EDB'*.*

DSSE security is captured in the real/ideal simulation paradigm with a leakage function family $\mathcal{L} = \{\mathcal{L}^{\mathsf{Stp}}, \mathcal{L}^{\mathsf{Srch}}, \mathcal{L}^{\mathsf{Updt}}\}$. $\mathcal{L}$ contains the leakage of setup, search, and update, with a sequence of historical operations as an implicit input.

Definition 2 (Adaptive Security of DSSE [26]) *A DSSE scheme is* $\mathcal{L}$*-adaptively-secure, if there exists a* PPT *simulator* $\mathcal{S}$ *such that* $|\Pr[\mathbf{Real}_{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda)$ *for any* PPT *adversary* $\mathcal{A}$*, where we define:*

$\mathbf{Real}_{\mathcal{A}}(1^\lambda)$*: The challenger executes* $\mathsf{Setup}(1^\lambda)$ *and sends (initially empty)* EDB *to* $\mathcal{A}$*.* $\mathcal{A}$ *adaptively makes a polynomial number of search queries with input* w *and update queries with input* (op, w, f)*. The challenger returns the transcripts generated by running* SrchTkn *on* w *or* UpdtTkn *on* (op, w, f)*. Finally,* $\mathcal{A}$ *returns a bit* b *output by the experiment.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^\lambda)$*:* $\mathcal{S}$ *generates (initially empty)* EDB *using* $\mathcal{L}^{\mathsf{Stp}}$ *and sends it to* $\mathcal{A}$*.* $\mathcal{A}$ *adaptively makes a polynomial number of search queries with input* w *and update queries with input* (op, w, f)*.* $\mathcal{S}$ *returns the transcripts for search queries (resp. update queries) using* $\mathcal{L}^{\mathsf{Srch}}(w)$ *(resp.* $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, f)$*). Eventually,* $\mathcal{A}$ *returns a bit* b *that is output by the experiment.*

Leakage Functions [26]. Let $\mathbb{O}$ be a sequence of DSSE operations issued so far. With u being the timestamp when an operation happens, $\mathbb{O}$ records (u, w) for a search on keyword w, or (u, op, w, f) for an update with (op, w, f). Typically,

- The *search pattern* sp indicates the repetition of searches on keywords. Formally, $\mathsf{sp}(w) = \{u | (u, w) \in \mathbb{O}\}$;
- The *update history* UpHist records all updates of keywords, *i.e.*, $\mathsf{UpHist}(w) = \{(u, \mathsf{op}, f) | (u, \mathsf{op}, w, f) \in \mathbb{O}\}$.

Intuitively, forward privacy means any revelation of secret knowledge (search token for a keyword w here) cannot be used to enable its associated function in the future (returning a subsequently updated keyword-file pair (w, f)). Bost [10] restricts the definition to consider only the update leakage free from w while it is fine to leak everything about f. Its generalization [29] aims to break the linkage between w and f. In particular, if update leakage has no information about f, prior search tokens cannot return results with f either, allowing relaxations that still capture the spirit of forward privacy.

Partial leakage of w while hiding f is an option. We limit the update leakage of (op, w, f) at u to $(\mathsf{op}, \mathsf{UpHist}_{<u}(w) \stackrel{?}{=} \emptyset)$, where $\mathsf{UpHist}_{<u}(w) := \mathsf{UpHist}(w) \setminus \{(u, \mathsf{op}, f)\}$, *i.e.*, the update history excluding the current update timestamped at u.

Definition 3 (Forward Privacy of DSSE [10, 29]) *We say an* $\mathcal{L}$*-adaptively-secure* DSSE *scheme is forward private if its update leakage* $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, f)$ *can be written as* $\mathcal{L}'(\mathsf{op}, f)$ *or* $\mathcal{L}''(\mathsf{op}, \mathsf{UpHist}_{<u}(w) \stackrel{?}{=} \emptyset)$ *for stateless functions* $\mathcal{L}', \mathcal{L}''$*.*

3 Hybrid Searchable Encryption

3.1 Syntax and System Model

Hybrid searchable encryption (HSE) considers three parties:

- a reader with the master secret key of HSE;
- multiple writers, each independently establishing a DSSE instance and an encrypted token set for the reader by generating the secret key and state for him/herself;
- a server storing encrypted databases and token sets.

Our formulation predefines a set of classes and assigns each writer to a different class[1]. We also use the *class* to identify a specific writer. HSE lets the reader do keyword searches over DSSE databases[2], with a *chosen subset* of writer classes each time. Each writer could update (and search) his/her DSSE instance while the reader enjoys sublinear searches.

Definition 4 (Hybrid Searchable Encryption (HSE)) *An* HSE *scheme consists of the following* PPT *algorithms:*

$(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{RSetup}(1^\lambda, n)$*: executed by the reader taking the input of security parameter* λ *and the number of writer classes* n*. It outputs a public/master secret key pair* $(\mathsf{pk}, \mathsf{msk})$*.*

$(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i, \mathsf{ETkn}_i) \leftarrow \mathsf{WSetup}(1^\lambda, i)$*: executed by a writer inputting its class* $i \in [n]$*. It outputs a secret key* k_i *and a secret state* st_i *for the writer to store locally and (initially empty) encrypted database* EDB_i *and encrypted token set* ETkn_i*.*

st_i *will be used in* $\mathsf{UpdtTkn}()$ *for creating update tokens. We let* $\mathsf{EDB} = \{\mathsf{EDB}_i\}_{i \in [n]}$ *and* $\mathsf{ETkn} = \{\mathsf{ETkn}_i\}_{i \in [n]}$*.* ETkn *collects* DSSE *search tokens for the reader/server, which will also be updated by* $\mathsf{Updt}()$*.* ETkn *can be viewed as part of* EDB*. We separate it for our description convenience.*

$\mathfrak{s} \leftarrow \mathsf{SrchTkn}(\mathsf{msk}, S, w)$*: executed by the reader taking as input a master secret key* msk*, a set* $S \subseteq [n]$ *of classes, and a keyword* w*. It outputs a search token* $\mathfrak{s}$*.*

$(\mathcal{R}, \mathsf{EDB}', \mathsf{ETkn}') \leftarrow \mathsf{Srch}(\mathfrak{s}, S, \mathsf{EDB}, \mathsf{ETkn})$*: executed by the server taking as input a search token* $\mathfrak{s}$*, a set* $S \subseteq [n]$ *of classes, an encrypted database* EDB*, and an encrypted token set* ETkn*. It outputs the search result* $\mathcal{R}$*, and (possibly) updated encrypted database* EDB' *and token set* ETkn'*.*

$(\mathfrak{u}, \mathsf{st}_i') \leftarrow \mathsf{UpdtTkn}(\mathsf{pk}, \mathsf{st}_i, i, \mathsf{k}_i, \mathsf{op}, w, f)$*: executed by the writer taking as input the public key* pk *of the reader, a state* st_i*, a class* $i \in [n]$*, the secret key* k_i*, an operation* $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}\}$*, and a keyword-file pair* (w, f) *to be updated. It outputs an update token* $\mathfrak{u}$ *and a new state* st_i'*.*

$(\mathsf{EDB}', \mathsf{ETkn}') \leftarrow \mathsf{Updt}(\mathfrak{u}, \mathsf{EDB}, \mathsf{ETkn})$*: executed by the server taking as input an update token* $\mathfrak{u}$*, an encrypted database* EDB*, and an encrypted token set* ETkn*. It updates the latter two inputs into* EDB' *and* ETkn' *and outputs them.*

Correctness. For all parameters λ and n, all $(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{RSetup}(1^\lambda, n)$, all $(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i, \mathsf{ETkn}_i) \leftarrow \mathsf{WSetup}(1^\lambda, i)$,

[1] Alternatively, the number of classes might be larger than that of writers.

[2] DSSE typically returns the identifiers of encrypted files matching the keyword. In HSE, the writers could encrypt the files under the reader's PKE.

$i \in [n]$, and all sequences of $\mathsf{Srch}()$ and $\mathsf{Updt}()$ over EDB using tokens generated respectively from $\mathsf{SrchTkn}(\mathsf{msk}, S, w)$ and $\mathsf{UpdtTkn}(\mathsf{pk}, \mathsf{st}_i, i, \mathsf{k}_i, \mathsf{op}, w, f)$, $i \in [n]$, Srch returns the correct results according to the inputs (i, op, w, f) of $\mathsf{UpdtTkn}$ when $i \in S$, except with negligible probability in λ.

Following features get HSE ready for real-world usage.

Search Efficiency. Search only involves the number of matching files and necessary traversals, sublinear in database sizes.

Update Efficiency. Update time of inserting or deleting a single keyword-file pair via $\mathsf{UpdtTkn}()$ and $\mathsf{Updt}()$ is constant.

Non-Interactiveness. No extra roundtrip between any parties is needed, except one inevitable roundtrip for the reader to authorize the server to search and return results.

Compact Tokens for Confined Search. The reader could choose an arbitrary writer subset to be searched at will via $\mathsf{SrchTkn}()$, which produces $O(1)$-size tokens for all subsets.

3.2 Threat Model

Our model considers an adversary who could corrupt any participants in HSE, except the *honest* reader. As a default corrupted party in searchable encryption, the server is *semi-honest*, who follows algorithm specifications to provide reliable storage (for encrypted databases) and services (for updates and searches) yet tries to derive sensitive information.

Note that the adversary has no others to collude with in DSSE or gains no advantage in colluding with the writers in PKSE since they have no secret. For our multi-writer setting, the adversary may corrupt some writers by getting their secret keys and secret states and instigate them to act *maliciously*.

HSE protects remaining entities, *i.e.*, the *honest* reader and *non-corrupted* writers, by preventing the adversary from learning beyond what it is supposed to know (to be formulated as leakage functions). Result authenticity is not in our scope.

3.3 General Security Definition

An HSE adversary can issue a sequence of oracle queries of three kinds: 1) corruption query, which returns the secret key and the secret state of a specific writer; 2) search query, which returns the search token of a specified keyword under a chosen writer subset; and 3) update query which returns the update token of a specified update tuple from certain writers. The adversary could issue queries depending on prior outcomes.

To define security, we extend the notion of *history*, a sequence of queries ever issued by the adversary, from SSE [19].

Definition 5 (History) *A history of* HSE *is a sequence of queries* $\mathcal{H} = \{\mathsf{Hist}_u\}_u$*, where sequence number* u *denotes the timestamp when the query happens and each* Hist_u *is in the form of either* (Corr, i)*,* (Srch, w, S)*, or* $(\mathsf{Updt}, i, \mathsf{op}, w, f)$*.*

Similar to SSE [19, 26], we introduce a leakage function family $\mathcal{L}_{\mathcal{H}} = \{\mathcal{L}^{\mathsf{Stp}}_{\mathcal{H}}, \mathcal{L}^{\mathsf{Srch}}_{\mathcal{H}}, \mathcal{L}^{\mathsf{Updt}}_{\mathcal{H}}, \mathcal{L}^{\mathsf{Corr}}_{\mathcal{H}}\}$ to control exactly the information of history $\mathcal{H}$ leaked during setup, search, update, and corruption, respectively. When an oracle is queried for the u-th operation, any function in $\mathcal{L}_{\mathcal{H}}$ is instantiated with $\mathcal{H}$ being the history consisting of the first $(u-1)$ operations and with the u-th operation as a function input[3]. It pinpoints 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.*, $\mathcal{H} = \emptyset$), $\mathcal{L} = \mathcal{L}^{\mathsf{Stp}}$.

We assume there exist HSE histories that are *non-singular*. It requires the existence of at least one other history with the same leakage. The assumption is necessary for searchable encryption [19], or all information of the history would leak.

Definition 6 (Non-Singular History [19]) *A history* $\mathcal{H}$ *is non-singular if there exists* $\mathcal{H}' \neq \mathcal{H}$ *where* $\mathcal{L}_{\mathcal{H}} = \mathcal{L}_{\mathcal{H}'}$ *that can be found in* PPT *given* $\mathcal{L}_{\mathcal{H}}$*.*

Definition 7 (Adaptive Security of HSE**)** *For all* PPT *adversary* $\mathcal{A}$ *and the game* $\mathsf{IND}^b_{\mathsf{HSE},\mathcal{A},\mathcal{L}}(1^\lambda)$ *in Figure 1,* HSE *is* $\mathcal{L}$*-adaptively-secure if the following quantity is negligible:* $\left|\Pr[\mathsf{IND}^0_{\mathsf{HSE},\mathcal{A},\mathcal{L}}(1^\lambda) = 1] - \Pr[\mathsf{IND}^1_{\mathsf{HSE},\mathcal{A},\mathcal{L}}(1^\lambda) = 1]\right|$.

Corruption Leakage. We define $I_c = \{i | (\mathsf{Corr}, i) \in \mathcal{H}\}$ as the set of corrupted writers. To capture the corruption leakage, for any class $i \in [n]$, we introduce a function $\mathsf{UpdtBy}(i)$ based on history $\mathcal{H}$, which lists all updates by i in the history. Formally, $\mathsf{UpdtBy}(i) = \{\mathsf{Hist}_u | \mathsf{Hist}_u = (\mathsf{Updt}, i, \mathsf{op}, w, f) \in \mathcal{H}\}$.

3.4 Epoch-Based Forward Privacy

We propose epoch-based forward privacy of HSE, inspired by forward-private PKE [12] and PKSE [40]. Commonly in these notions, each encryptor/writer works independently. Security definitions for PKSE do not feature parameterized leakage, let alone forward-private PKE. In contrast, leakages play an important role in HSE – The definition of forward privacy depends on the definition of update leakage. For security proof, simulation of different oracles crucially depends on the corresponding leakages. They thus precisely capture what are the leakages when an HSE scheme is used in practice.

Consider the system timeline as a set of timestamps. We divide it into sequential fragments, each assigned with an epoch sequence number e. We say an HSE operation happens at epoch e if the timestamp when it is executed belongs to e. All algorithms implicitly take e as an input.

To model the effects of temporal variation in HSE on security, we introduce $\mathsf{Epoch}O$, which monotonically increases the global epoch e whenever invoked. It enables the adversary to decide how its historical operations are arranged at epochs.

We update our HSE security game with $\mathsf{Epoch}O$ as Figure 1. $\mathsf{Corr}O$, $\mathsf{Srch}O$, and $\mathsf{Updt}O$ are mostly unchanged, except that the epoch information is attached when extending the

[3] For the u-th operation, a subtlety is that HSE evaluates the leakage before including it into $\mathcal{H}$, while DSSE considers it being put into $\mathbb{O}$ immediately.

$\mathsf{IND}^b_{\mathsf{HSE},\mathcal{A},\mathcal{L}}(1^\lambda)$

$(n, \mathsf{st}_\mathcal{A}) \leftarrow \mathcal{A}(1^\lambda)$

$(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{RSetup}(1^\lambda, n)$

$\forall i \in [n], (\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i, \mathsf{ETkn}_i) \leftarrow \mathsf{WSetup}(1^\lambda, i)$

$\mathsf{EDB} := \{\mathsf{EDB}_i\}_{i\in[n]}, \mathsf{ETkn} := \{\mathsf{ETkn}_i\}_{i\in[n]}$

$\mathcal{H}_0 := \mathcal{H}_1 := \emptyset$; $\boxed{e := 0}$

$\mathcal{O} = \{\mathsf{Corr}\mathcal{O}_b, \mathsf{Srch}\mathcal{O}_b, \mathsf{Updt}\mathcal{O}_b, \boxed{\mathsf{Epoch}\mathcal{O}}\}$

$b' \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{st}_\mathcal{A}, \mathsf{pk}, \mathsf{EDB}, \mathsf{ETkn})$

return b'

$\mathsf{Srch}\mathcal{O}_b(\{S_j, w_j\}_{j\in\{0,1\}})$

if $\mathcal{L}^{\mathsf{Srch}}_{\mathcal{H}_0}(w_0, S_0) = \mathcal{L}^{\mathsf{Srch}}_{\mathcal{H}_1}(w_1, S_1)$

$\forall j \in \{0,1\}, \mathcal{H}_j := \mathcal{H}_j || (\mathsf{Srch}, w_j, S_j)$

return $\mathsf{SrchTkn}(\mathsf{msk}, S_b, w_b)$

else return $\perp$

$\mathsf{Updt}\mathcal{O}_b(\{i_j, \mathsf{op}_j, w_j, f_j\}_{j\in\{0,1\}})$

if $\mathcal{L}^{\mathsf{Updt}}_{\mathcal{H}_0}(i_0, \mathsf{op}_0, w_0, f_0) = \mathcal{L}^{\mathsf{Updt}}_{\mathcal{H}_1}(i_1, \mathsf{op}_1, w_1, f_1)$

$\forall j \in \{0,1\}, \mathcal{H}_j := \mathcal{H}_j || (\mathsf{Updt}, i_j, \mathsf{op}_j, w_j, f_j)$

return $\mathsf{UpdtTkn}(\mathsf{pk}, \mathsf{st}_{i_b}, i_b, \mathsf{k}_{i_b}, \mathsf{op}_b, w_b, f_b)$

else return $\perp$

$\mathsf{Corr}\mathcal{O}_b(i_0, i_1)$

if $\mathcal{L}^{\mathsf{Corr}}_{\mathcal{H}_0}(i_0) = \mathcal{L}^{\mathsf{Corr}}_{\mathcal{H}_1}(i_1)$

$\forall j \in \{0,1\}, \mathcal{H}_j := \mathcal{H}_j || (\mathsf{Corr}, i_j)$

return $(\mathsf{k}_{i_b}, \mathsf{st}_{i_b})$

else return $\perp$

$\boxed{\underline{\mathsf{Epoch}\mathcal{O}()}\ e := e+1}$

Figure 1: Security Game for HSE (Boxed Codes for Epoch-Based Notions in Sections 3.4 and 6)

history, which can be inferred from the query timestamp. Each component Hist_u of history $\mathcal{H}$ is in the form of (Corr, i, e), (Srch, w, S, e), or $(\mathsf{Updt}, i, \mathsf{op}, w, f, e)$, where e is the epoch to which the query timestamp u belongs. Note that the notion of history now implicitly defines the current epoch.

We define two auxiliary functions based on $\mathcal{H}$:

- $W_{\mathsf{Srch}}(i, e)$ reports the set of keywords that has been searched over any subset containing i during epoch e. Formally, $W_{\mathsf{Srch}}(i, e) = \{w | (\mathsf{Srch}, w, S, e) \in \mathcal{H} \wedge i \in S\}$;
- $\mathsf{UpHist}(i, w)$ reports the *update history* of keyword w by i so far. Formally, $\mathsf{UpHist}(i, w) = \{(u, \mathsf{op}, f) | \mathsf{Hist}_u = (\mathsf{Updt}, i, \mathsf{op}, w, f) \in \mathcal{H}\}$.

Epoch-based forward privacy requires that the server cannot learn whether the updated file from a non-corrupted writer i matches a keyword w that may have been searched in prior epochs but not yet at the current one. Unlike the regular notion, if w has been searched for over any writer subset containing i at epoch e, *i.e.*, $w \in W_{\mathsf{Srch}}(i, e)$, any subsequent update from i within epoch e could still be searchable. It implies the leakage of a bit, indicating whether the updated keyword has been searched at the same epoch. Correspondingly, update leakages can have different forms. If an epoch is infinitesimal, epoch-based forward privacy degenerates to the regular one.

Definition 8 (Forward Privacy of HSE) *An $\mathcal{L}$-adaptively-secure HSE is forward-private if the update leakage $\mathcal{L}^{\mathsf{Updt}}(i, \mathsf{op}, w, f)$ of any update (op, w, f) by any writer class $i \notin I_c$ that happens at its respective epoch e can be written as either $\mathcal{L}'(i, \mathsf{op}, f)$ or $\mathcal{L}''(i, \mathsf{op}, \mathsf{UpHist}(i, w) \stackrel{?}{=} \emptyset)$, provided $w \notin W_{\mathsf{Srch}}(i, e)$, where $\mathcal{L}', \mathcal{L}''$ are stateless functions.*

The update leakage includes a bit $(\mathsf{UpHist}(i, w) \stackrel{?}{=} \emptyset)$, indicating whether the keyword has already been updated by the writer, which allows for more efficient HSE instantiations.

3.5 Generic Construction

Figure 2 details our generic HSE construction (G-HSE) from DSSE with non-stateful deterministic search tokens (*e.g.*, [26]) and anonymous IBE. We refer to prior works [2, 7] for IND-CPA/ANON security of IBE and its transformation to PKSE. In G-HSE, the reader uses IBE.Setup() to set up n IBE instances for n writers. Each writer independently sets up a DSSE instance via DSSE.Setup().

For an update of keyword w, apart from creating a DSSE update token and sending it to the server, the writer encrypts the DSSE search token of w via IBE.Enc() using w as the identity if w has never been updated before. To indicate it, each writer state contains B_i as a $|\mathcal{W}|$-bit list for the keyword space $\mathcal{W}$ (1 for positive). The server collects these encrypted DSSE search tokens (decryptable by keys from IBE.Ext()).

For an HSE search query of w, the reader provides decryption keys for w under IBE instances of target writers.Obtaining DSSE search tokens by decrypting IBE ciphertexts from target writers, the server retrieves files matching w via DSSE.

G-HSE search first attempts to decrypt the encrypted token sets then performs the DSSE search. The search time is the sum of traversal time over the target token sets and DSSE search time, which is typically sublinear. Both update token size and complexity are constant. However, it is not forward private, and its search token size grows with the target subset.

Regarding security, beyond DSSE update leakage, a G-HSE update leaks for efficiency benefits whether the keyword has been updated by the class. The DSSE search token is IBE-encrypted during a G-HSE update, which leaks nothing when IBE is IND-CPA- and IND-ANON-secure. For search, as G-HSE is a public-key scheme, it allows keyword-guessing attacks[4]. G-HSE search leakage also contains DSSE search leakage. We summarize G-HSE leakage functions: $\mathcal{L}^{\mathsf{Stp}}_{\mathsf{hse}}(n) = \{i, \mathcal{L}^{\mathsf{Stp}}_{\mathsf{sse},i}\}_{i\in[n]}$, $\mathcal{L}^{\mathsf{Corr}}_{\mathsf{hse}}(i) = \{\mathsf{UpdtBy}(i)\}$, $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{hse}}(w, S) = \{i, \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w), w\}_{i\in S}$, and $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{hse}}(i, \mathsf{op}, w, f) =$

$$\begin{cases} \{i, \mathsf{op}, w, f\} & \text{if } i \in I_c, \\ \{i, \mathsf{UpHist}(i, w) \stackrel{?}{=} \emptyset, \mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}(\mathsf{op}, w, f)\} & \text{otherwise.} \end{cases}$$

Theorem 1 *Assuming that the underlying IBE is IND-CPA-secure and IND-ANON-secure, and the underlying DSSE is*

[4] One could use a function-private IBE scheme [9], which prevents leakage of the ID "beyond the absolute minimum." It requires super-logarithmic min-entropy as in the notion of anonymous ciphertext indistinguishability [16].

$\mathsf{RSetup}(1^\lambda, n)$

foreach $i \in [n]$
  $(\mathsf{pk}_i, \mathsf{msk}_i) \leftarrow \mathsf{IBE.Setup}(1^\lambda)$
$\mathsf{pk} := \{\mathsf{pk}_i\}_{i\in[n]}$
$\mathsf{msk} := \{\mathsf{msk}_i\}_{i\in[n]}$
return $(\mathsf{pk}, \mathsf{msk})$

$\mathsf{UpdtTkn}(\mathsf{pk}, \mathsf{st}_i, i, \mathsf{k}_i, \mathsf{op}, w, f)$

parse $\mathsf{st}_i = (B_i, \ldots)$
$(\mathsf{u}_{\mathsf{sse}}, \mathsf{st}'_i)$
  $\leftarrow \mathsf{DSSE.UpdtTkn}$
    $(\mathsf{st}_i, \mathsf{k}_i, \mathsf{op}, w, f)$
$\mathsf{st}_i := \mathsf{st}'_i$
if $B_i[w] = 0$
  $(\mathfrak{s}_{\mathsf{sse}}, \mathsf{st}'_i)$
    $\leftarrow \mathsf{DSSE.SrchTkn}(\mathsf{st}_i, \mathsf{k}_i, w)$
  $c \leftarrow \mathsf{IBE.Enc}(\mathsf{pk}_i, w, \mathfrak{s}_{\mathsf{sse}})$
  $B_i[w] := 1$
else $c := \perp$
$\mathsf{u} = \{i, c, \mathsf{u}_{\mathsf{sse}}\}$
return $(\mathsf{u}, \mathsf{st}'_i)$

$\mathsf{Updt}(\mathsf{u} = (i, c, \mathsf{u}_{\mathsf{sse}}), \mathsf{EDB}, \mathsf{ETkn})$

if $c \neq \perp$ **then** $\mathsf{ETkn}'_i := \mathsf{ETkn}_i \cup \{c\}$
$\mathsf{EDB}'_i \leftarrow \mathsf{DSSE.Updt}(\mathsf{u}_{\mathsf{sse}}, \mathsf{EDB}_i)$
return $(\mathsf{EDB}', \mathsf{ETkn}')$

$\mathsf{WSetup}(1^\lambda, i)$

$(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i)$
  $\leftarrow \mathsf{DSSE.Setup}(1^\lambda)$
$\mathsf{ETkn}_i := \emptyset$
return $(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i, \mathsf{ETkn}_i)$

$\mathfrak{s} \leftarrow \mathsf{SrchTkn}(\mathsf{msk}, S, w)$

foreach $i \in S$
  $\mathsf{dk}_i \leftarrow \mathsf{IBE.Ext}(\mathsf{msk}_i, w)$
$\mathfrak{s} := \{i, \mathsf{dk}_i\}_{i\in S}$
return $\mathfrak{s}$

$\mathsf{Srch}(\mathfrak{s}, S, \mathsf{EDB}, \mathsf{ETkn})$

parse $\mathfrak{s} = \{i, \mathsf{dk}_i\}_{i\in S}$
parse $\mathsf{EDB} = \{\mathsf{EDB}_i\}_{i\in[n]}$
parse $\mathsf{ETkn} = \{\mathsf{ETkn}_i\}_{i\in[n]}$
$\mathcal{R} := \emptyset$
foreach $i \in S$
  foreach $c \in \{\mathsf{ETkn}_i\}$
    $\mathfrak{s}_{\mathsf{sse}} \leftarrow \mathsf{IBE.Dec}(\mathsf{dk}_i, c)$
    if $\mathfrak{s}_{\mathsf{sse}} \neq \perp$,
      (R, EDB'_i)
        $\leftarrow \mathsf{DSSE.Srch}$
          $(\mathfrak{s}_{\mathsf{sse}}, \mathsf{EDB}_i)$
      $\mathcal{R} := \mathcal{R} \cup R$
return $(\mathcal{R}, \mathsf{EDB}', \mathsf{ETkn})$

Figure 2: G-HSE from DSSE and IBE

$\mathcal{L}_{\mathsf{sse}}$-adaptively-secure, G-HSE *is $\mathcal{L}_{\mathsf{hse}}$-adaptively-secure.*

We omit its proof as it is subsumed by that of FP-HSE.

4 ID-Coupling Key-Aggregate Encryption

We propose identity-coupling key-aggregate encryption (ICKAE), which equips an exponential space of identities to key-aggregate encryption (KAE) [18]. (IC)KAE ciphertexts are generated with respect to a class. Decryption keys for an arbitrarily chosen subset of classes can be aggregated, with both ciphertexts and aggregated keys being *compact*.

Syntax-wise, ICKAE upgrades KAE from PKE to IBE. Coupling ciphertexts and keys with identities enables more fine-grained and flexible access control. Moreover, we expect ICKAE can hide the identity from those who cannot decrypt, which unlocks new applications concerning anonymity.

Design-wise, we start from the original KAE scheme [18][5]. Trivially extending it to ICKAE only affords a loose security reduction with $O(1)$ key extraction oracle queries. We resolve the issue via the random-bit trick of Katz and Wang [27]. Moreover, we strive to use only two pairings in decryption[6].

4.1 Syntax and Security

In ICKAE, any writer can encrypt under a public key their messages for a specified class and an identity string. The private key holder can create an aggregated decryption key, which can decrypt ciphertexts with respect to the identity for any subset of classes. ICKAE solves the first downside of G-HSE. The data owner only needs to store a *single* key pair (vs. linear in the number of predefined writer classes), which can generate a *constant-size* aggregated key of several classes coupled with a specified identity. As we use identity as a keyword in HSE, we expect ICKAE to be identity-anonymous.

Definition 9 (ID-Coupling Key-Aggregate Encryption) *An* ICKAE *scheme consists of the following* PPT *algorithms:*

$\mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda, n)$ *takes as input a security parameter* λ *and the number of classes* n*. It outputs the parameter* param*, an implicit input for other algorithms.*

$(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{KeyGen}()$ *outputs a key pair* $(\mathsf{pk}, \mathsf{msk})$*.*

$c \leftarrow \mathsf{Enc}(\mathsf{pk}, i, \mathsf{id}, m)$ *takes as input a public key* pk*, a class* $i \in [n]$*, an identity* id*, and a message* m*. It outputs a ciphertext* c*.*

$\mathsf{ak} \leftarrow \mathsf{Ext}(\mathsf{msk}, S, \mathsf{id})$ *inputs a master secret key* msk*, a set* $S \subseteq [n]$*, and an identity* id*. It outputs an aggregated key* ak*.*

$m \leftarrow \mathsf{Dec}(\mathsf{ak}, S, i, c)$ *takes an aggregated key* ak*, a class set* $S \subseteq [n]$*, and a class* $i \in [n]$ *of a ciphertext* c*. It decrypts* c *to* m*.*

Correctness. For any integers λ, n, any $S \subseteq [n]$, $i \in S$, id, and m, $\Pr[\mathsf{Dec}(\mathsf{ak}, S, i, c) = m : \mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda, n), (\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{KeyGen}(), c \leftarrow \mathsf{Enc}(\mathsf{pk}, i, \mathsf{id}, m), \mathsf{ak} \leftarrow \mathsf{Ext}(\mathsf{msk}, S, \mathsf{id})] = 1$.

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

Confidentiality and Anonymity. In the confidentiality game, the adversary is asked to distinguish a ciphertext of one of its chosen messages under its specified identity, while the anonymity game challenges the adversary with the ciphertext of its chosen message under one of two identities it specifies.

Definition 10 ICKAE *is* X*-secure if for any* PPT $\mathcal{A}$*,*

$$\left|\Pr[\mathsf{X}^0_{\mathsf{ICKAE},\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathsf{X}^1_{\mathsf{ICKAE},\mathcal{A}}(1^\lambda) = 1]\right| \leq \mathsf{negl}(\lambda)$$

where the games $\mathsf{X}^b_{\mathsf{ICKAE},\mathcal{A}}(1^\lambda)$ *are defined in Figure 3 with* $\mathsf{X} \in \{\mathsf{IND\text{-}CPA}, \mathsf{IND\text{-}ANON}\}$.

We weaken the flexibility of key extraction oracle $\mathsf{Ext}O$ for a more efficient instantiation: it allows only one adversarial choice of subset S for each id. In our HSE application, id also serves as an epoch, enforcing a per-epoch search scope.

[5] It uses symmetric pairing but can be easily ported to asymmetric pairing.

[6] A side effect is that the aggregate (decryption) key generation is "almost" deterministic. Roughly, introducing a random factor from $\mathbb{Z}_p$ often leads to one more pairing in decryption. Deterministic key generation is problematic when there are two levels of secret, which affects the key extraction flexibility.

$\mathsf{isChallenge}(S,\mathsf{id})$

if $(i^* \in S)$
  if $(\mathsf{id} = \mathsf{id}^*)$ **return** true
  if $(\mathsf{id} \in \{\mathsf{id}_0^*, \mathsf{id}_1^*\})$ **return** true
return false

$\mathsf{Ext}O(S,\mathsf{id})$

if $(\mathsf{id} \in idSet)$ **return** $\perp$
if $(\mathsf{isChallenge}(S,\mathsf{id}))$ **return** $\perp$
$idSet := idSet \cup \{\mathsf{id}\}$
return $\mathsf{ak} := \mathsf{Ext}(\mathsf{msk}, S, \mathsf{id})$

$\mathsf{IND\text{-}CPA}^b_{\mathsf{ICKAE},\mathcal{A}}(1^\lambda)$

$idSet := \emptyset$; $i^* := \mathsf{id}^* := \perp$
$\mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda, n)$
$(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{KeyGen}()$
$(\mathsf{st}, m_0, m_1, i^*, \mathsf{id}^*) \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{pk})$
$c^* \leftarrow \mathsf{Enc}(\mathsf{pk}, i^*, \mathsf{id}^*, m_b)$
if $(\mathsf{isChallenge}(\{i^*\}, \mathsf{id}^*))$
  return $\perp$
return $b' \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{st}, c^*)$

$\mathsf{IND\text{-}ANON}^b_{\mathsf{ICKAE},\mathcal{A}}(1^\lambda)$

$idSet := \emptyset; i^* := \mathsf{id}_0^* := \mathsf{id}_1^* := \perp$
$\mathsf{param} \leftarrow \mathsf{Setup}(1^\lambda, n)$
$(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{KeyGen}()$
$(\mathsf{st}, m, i^*, \mathsf{id}_0^*, \mathsf{id}_1^*) \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{pk})$
$c^* \leftarrow \mathsf{Enc}(\mathsf{pk}, i^*, \mathsf{id}_b^*, m)$
if $(\mathsf{isChallenge}(\{i^*\}, \mathsf{id}_b^*))$
  return $\perp$
return $b' \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{st}, c^*)$

Figure 3: IND-CPA & IND-ANON Security of ICKAE

4.2 Construction

We construct an ICKAE scheme over a tuple of cyclic groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t)$ each of prime order p and equipped with pairing $e : \mathbb{G}_1 \times \mathbb{G}_2$ to $\mathbb{G}_t$. We use the implicit notion for group elements [21]. Fix arbitrary generators $[1]_1 \in \mathbb{G}_1$ and $[1]_2 \in \mathbb{G}_2$, we define $[1]_t := e([1]_1, [1]_2)$ as a generator of $\mathbb{G}_t$. For $i \in \{1,2,t\}$ and $x \in \mathbb{Z}_p$, $[x]_i \in \mathbb{G}_i$ denotes the group element whose discrete logarithm base $[1]_i$ is x. Group operations are written additively, *i.e.*, $[x]_i + [y]_i := [x+y]_i \in \mathbb{G}_i$. The pairing is written multiplicatively, *i.e.*, $[x]_1[y]_2 := e([x]_1, [y]_2) = [xy]_t$.

Let $H : \{0,1\}^* \to \mathbb{G}_1$ and $G : \mathbb{G}_t \to \{0,1\}^\lambda$ be two cryptographic hash functions. Figure 4 presents our construction. Its correctness can be checked by noting that $[u]_t$ equals

$$\begin{aligned}
&\sum_{j\in S}[\alpha^{n+1-j}]_1[c_2]_2 - ([k]_1 + \sum_{j\in S\setminus\{i\}}[\alpha^{n+1+i-j}]_1)[c_1]_2 \\
=& r\sum_{j\in S}[\alpha^{n+1-j}]_1([\gamma]_2 + [\alpha^i]_2) \\
&-(\gamma\sum_{j\in S}[\alpha^{n+1-j}]_1 + \delta[h_b]_1 + \sum_{j\in S\setminus\{i\}}[\alpha^{n+1+i-j}]_1)[r]_2 \\
=& -r[h_b]_1[\delta]_2 + r[\alpha^{n+1}]_t.
\end{aligned}$$

For security, $\mathsf{Ext}()$ should only be executed at most once for each id with the same random choice of b. In practice, b can be generated from a pseudorandom function taking id as an input. Theorem 2 asserts the confidentiality of our ICKAE, with its proof in Appendix A. Anonymity can be reduced to the bilinear Diffie-Hellman assumption $(([h]_1, [r]_2, [\delta]_2, [hr\delta]_t))$ in a rather direct manner. For computational consistency, it largely follows an existing one [2] for the original PKSE scheme [7]. Besides, a generic upgrade for consistency exists [2].

Theorem 2 *Our* ICKAE *is* IND-CPA*-secure under the n-BDHE assumption (e.g., [18]) in the random oracle model.*

$\mathsf{Setup}(1^\lambda, n)$

$\alpha \leftarrow\$ \mathbb{Z}_p$
$\mathbb{I} := i \in [2n] \setminus \{n+1\}$
return $(\{[\alpha^i]_1\}_{\mathbb{I}}, \{[\alpha^i]_2\}_{\mathbb{I}}, [\alpha^{n+1}]_t)$

$\mathsf{KeyGen}()$

$\gamma, \delta \leftarrow\$ \mathbb{Z}_p$
$\mathsf{msk} := (\gamma, \delta)$
$\mathsf{pk} := ([\gamma]_2, [\delta]_2)$
return $(\mathsf{pk}, \mathsf{msk})$

$\mathsf{Ext}(\mathsf{msk}, S, \mathsf{id})$

$b \leftarrow\$ \{0,1\}$; $[h_b]_1 := H(\mathsf{id}, b)$
$[k]_1 := \gamma\sum_{j\in S}[\alpha^{n+1-j}]_1 + \delta[h_b]_1$
return $\mathsf{ak} := ([k]_1, b)$

$\mathsf{Enc}(\mathsf{pk}, i, \mathsf{id}, m)$

$r \leftarrow\$ \mathbb{Z}_p$; $[c_1]_2 := [r]_2$
$[c_2]_2 := r([\gamma]_2 + [\alpha^i]_2)$
$[h_0]_1 := H(\mathsf{id}, 0)$; $[h_1]_1 := H(\mathsf{id}, 1)$
$c_{3,0} := m \oplus G(r[\alpha^{n+1}]_t - r[h_0]_1[\delta]_2)$
$c_{3,1} := m \oplus G(r[\alpha^{n+1}]_t - r[h_1]_1[\delta]_2)$
return $c := ([c_1]_2, [c_2]_2, c_{3,0}, c_{3,1})$

$\mathsf{Dec}(\mathsf{ak}, S, i, c)$

parse $\mathsf{ak} = ([k]_1, b)$, $c = ([c_1]_2, [c_2]_2, c_{3,0}, c_{3,1})$
$[u]_t := \sum_{j\in S}[\alpha^{n+1-j}]_1[c_2]_2 - ([k]_1 + \sum_{j\in S\setminus\{i\}}[\alpha^{n+1+i-j}]_1)[c_1]_2$
return $c_{3,b} \oplus G([u]_t)$

Figure 4: Construction of ICKAE

5 Epoch-Based Forward-Private DSSE

Recall that DSSE clients usually realize forward privacy by keeping a state per keyword and determining the address of the next update based on its state [10, 29]. Only until the *next search* on the keyword is issued will its state be revealed and then refreshed. A refreshed new state results in new addresses of its subsequent updates, thus achieving forward privacy.

As HSE expects no synchronous communication, the reader and any writer cannot share whether a state has been revealed (by the reader) and should be refreshed (by writers). To resolve this hurdle, we instead use the epoch derived from the system time as a constantly refreshing "state." Our epoch-based DSSE scheme E-DSSE enables a search token (for the current epoch) to (determine addresses of and) retrieve tuples updated/to-be-updated at the current epoch and all updates at prior epochs. This is realized by maintaining (implicit) links for updates inter- and intra-epochs. As usual, a search token has no link to any updates after its epoch.

Such a property follows the spirit of DSSE forward privacy. Its reliance on a loosely synchronized global clock makes the resulting DSSE schemes applicable to the HSE setting.

5.1 Definition

To formally define epoch-based forward privacy of DSSE, we introduce $\mathsf{Epoch}O$ for the adversary to advance the epoch (as in Section 3.4) and record the epoch information in the DSSE operation sequence $\mathbb{O}$: either $((u,e),w)$ for a search on keyword w or $((u,e),\mathsf{op},w,f)$ for an update with (op,w,f),

where u is the timestamp of an operation and e is the epoch to which u belongs. The search patterns sp and update history UpHist (Section 2) also record the epoch information. Moreover, $\mathsf{UpHist}_{<e}(w)$ and $\mathsf{UpHist}_{e}(w)$ denote all updates of keyword w before and at epoch e, respectively.

We further define the set of keywords being searched during epoch e by $W_{\mathsf{Srch}}(e) = \{w | ((u,e),w) \in \mathbb{O}\}$.

Epoch-based forward privacy confines the update leakage according to recent searches. If no search has been issued for the keyword to be updated at the same epoch, the update leakage stays the same as regular forward privacy (Definition 3).

Definition 11 (Epoch-Based Forward Privacy of DSSE) *An $\mathcal{L}$-adaptively-secure DSSE scheme is epoch-based forward-private if for any update (op, w, f) at epoch e with $w \notin W_{\mathsf{Srch}}(e)$, the update leakage function $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, f)$ is in the form of $\mathcal{L}'(\mathsf{op}, f)$ or $\mathcal{L}''(\mathsf{op}, \mathsf{UpHist}_{<u}(w) \stackrel{?}{=} \emptyset)$, where $\mathcal{L}', \mathcal{L}''$ are stateless functions.*

DSSE schemes satisfying Definition 3 are, by definition, epoch-based forward-private, but not vice versa. However, the new concept inspires epoch-based applications and potentially provides richer functionalities than existing schemes. Leakage of $\mathsf{UpHist}_{<u}(w) \stackrel{?}{=} \emptyset$ is motivated by efficiency concerns.

For our FP-HSE usage, DSSE tokens with the temporal search ability will be encrypted to the server. It deviates from the DSSE security model, where the server will immediately see and execute the search token. To capture this, we let the operation sequence $\mathbb{O}$ record $((u,e),\text{“}\mathsf{Tkn}\text{”},w)$ for the search token generation on w. We use $\mathcal{L}^{\mathsf{SrchTkn}}(w)$ to denote the leakage caused by issued yet unused search tokens and separate it from the search leakage $\mathcal{L}^{\mathsf{Srch}}(w)$. In the DSSE context, such search-token leakage captures the token delegations to readers who may reveal the token to the server at different times when the actual search should be executed.

5.2 Construction

We propose E-DSSE, the first DSSE scheme with a tunable tradeoff between forward privacy and flexibility of temporal delegations. A search token retrieves all results updated since the initialization (“inter-epoch” updates) and to be updated at the current epoch (“intra-epoch” updates), but not at future epochs. As a typical design, E-DSSE implicitly links update tuples for each keyword of the same epoch via pseudorandom addresses derived using the same secret but different counters.

To travel to the most recent epoch where an update of w happened (and avoid traversal over each “missing” epoch), $\mathsf{UpdtTkn}()$ keeps it as local state $\mathsf{T}_{\mathsf{ep}}[w]$. $\mathsf{SrchTkn}()$ always outputs both the search token $\mathfrak{s}_e$ for the current epoch e and an inter-epoch token $\mathfrak{s}_{\mathsf{T}_{\mathsf{ep}}[w]}$, which frees $\mathsf{SrchTkn}()$ from local state update. To keep $\mathsf{Srch}()$ mostly a single traversal loop for optimality and compatibility with token-based DSSE, $\mathfrak{s}_{\mathsf{T}_{\mathsf{ep}}[w]}$ is stored along with a dummy update of the current epoch (if no update exists) and picked up when the first update of each epoch (if it exists) is retrieved. $\mathsf{UpdtTkn}()$ will replace such a dummy with the actual first update while keeping $\mathfrak{s}_{\mathsf{T}_{\mathsf{ep}}[w]}$. Even two searches at different epochs (say e' and $e'+1$) may implant the same $\mathfrak{s}_{\mathsf{T}_{\mathsf{ep}}[w]}$, the earlier one ($e'$ in our example) will not be traversed as there was no actual update (which is why a search at epoch $e'+1$ implanted $\mathfrak{s}_{\mathsf{T}_{\mathsf{ep}}[w]}$ but not $\mathfrak{s}_{e'}$).

$\mathsf{Setup}(1^\lambda)$

$\mathsf{k} \leftarrow\$ \{0,1\}^\lambda$
$\mathsf{T}_{\mathsf{ct}}, \mathsf{T}_{\mathsf{ep}}, \mathsf{EDB} \leftarrow$ empty map
$\mathsf{st} := (\mathsf{T}_{\mathsf{ct}}, \mathsf{T}_{\mathsf{ep}})$
return $(\mathsf{k}, \mathsf{st}, \mathsf{EDB})$

$\mathsf{SrchTkn}(\mathsf{st}, \mathsf{k}, w)$

get current epoch e
parse $\mathsf{st} = (\mathsf{T}_{\mathsf{ct}}, \mathsf{T}_{\mathsf{ep}})$
$\mathfrak{s}_e := F(\mathsf{k}, w||e)$
$\mathsf{addr} := \mathsf{val} := \perp$
if $(\mathsf{T}_{\mathsf{ep}}[w] \neq e) \wedge (\mathsf{T}_{\mathsf{ep}}[w] \neq \perp)$
  $x := F(\mathsf{k}, w||\mathsf{T}_{\mathsf{ep}}[w])$
  $y := 0^{|\mathsf{op}||f|}$
  $\mathsf{addr} := H_1(\mathfrak{s}_e||1)$
  $\mathsf{val} := (y||x) \oplus H_2(\mathfrak{s}_e||1)$
  $\mathfrak{s} := (\mathfrak{s}_e, \mathsf{addr}, \mathsf{val})$
return $(\mathfrak{s}, \mathsf{st})$

$\mathsf{UpdtTkn}(\mathsf{st}, \mathsf{k}, \mathsf{op}, w, f)$

get current epoch e
parse $\mathsf{st} = (\mathsf{T}_{\mathsf{ct}}, \mathsf{T}_{\mathsf{ep}})$
if $(\mathsf{T}_{\mathsf{ct}}[w] = \perp) \vee (\mathsf{T}_{\mathsf{ep}}[w] \neq e)$
  $\mathsf{T}_{\mathsf{ct}}[w] := 0$
$x := 0^\lambda; \mathsf{T}_{\mathsf{ct}}[w] := \mathsf{T}_{\mathsf{ct}}[w] + 1$
if $(\mathsf{T}_{\mathsf{ct}}[w] = 1) \wedge (\mathsf{T}_{\mathsf{ep}}[w] \neq \perp)$
  $x := F(\mathsf{k}, w||\mathsf{T}_{\mathsf{ep}}[w])$
$\mathsf{T}_{\mathsf{ep}}[w] := e$
$K := F(\mathsf{k}, w||e)||\mathsf{T}_{\mathsf{ct}}[w]$
$\mathsf{addr} := H_1(K)$
$\mathsf{val} := (\mathsf{op}||f||x) \oplus H_2(K)$
$\mathsf{u} := (\mathsf{addr}, \mathsf{val})$
return $(\mathsf{u}, \mathsf{st}' := (\mathsf{T}_{\mathsf{ct}}, \mathsf{T}_{\mathsf{ep}}))$

$\mathsf{Updt}((\mathsf{addr}, \mathsf{val}), \mathsf{EDB})$

$\mathsf{EDB}[\mathsf{addr}] := \mathsf{val}$
return $\mathsf{EDB}' := \mathsf{EDB}$

$\mathsf{Srch}(\mathfrak{s}, \mathsf{EDB})$

parse $\mathfrak{s} = (\mathfrak{s}_e, \mathsf{addr}, \mathsf{val}); R := \emptyset$
if $(((\mathsf{addr}, \mathsf{val}) \neq (\perp, \perp)) \wedge (\mathsf{EDB}[\mathsf{addr}] = \perp))$ $\mathsf{EDB}[\mathsf{addr}] := \mathsf{val}$
while $\mathfrak{s}_e \neq 0^\lambda$
  $\mathsf{ct} := 1$; $\mathsf{addr} := H_1(\mathfrak{s}_e||\mathsf{ct})$; $\mathfrak{s}'_e := 0^\lambda$
  while $\mathsf{EDB}[\mathsf{addr}] \neq \perp$
    $\mathsf{op}||f||x := \mathsf{EDB}[\mathsf{addr}] \oplus H_2(\mathfrak{s}_e||\mathsf{ct})$
    if $\mathsf{ct} = 1$ **then** $\mathfrak{s}'_e := x$
    if $\mathsf{op} = \mathsf{add}$ **then** $R := R \cup \{f\}$ **else** $R := R \setminus \{f\}$
    $\mathsf{ct} := \mathsf{ct} + 1$; $\mathsf{addr} := H_1(\mathfrak{s}_e||\mathsf{ct})$
  $\mathfrak{s}_e := \mathfrak{s}'_e$
return R

Figure 5: E-DSSE: Epoch-Based Forward-Private DSSE

Figure 5 details our E-DSSE scheme, where H_1 and H_2 are cryptographic hash functions, and F is a pseudorandom function family (PRF) with λ-bit outputs. We assume *retrieving an empty entry* from a map (*e.g.*, T_{ct} or T_{ep}) returns $\perp$.

Setup. Let k be a λ-bit key for F. T_{ep} and T_{ct} are initially empty maps. $\mathsf{T}_{\mathsf{ep}}[w]$ stores the epoch when the last update of w happens, and $\mathsf{T}_{\mathsf{ct}}[w]$ stores the update counter of w at epoch $\mathsf{T}_{\mathsf{ep}}[w]$. An initially empty map for EDB is outsourced.

Search. The client computes search token $\mathfrak{s}_e$ for keyword w

and current epoch e as $F(\mathsf{k}, w||e)$. If w has not been updated at epoch e ($\mathsf{T}_{\mathsf{ep}}[w] \neq e$) but updated during previous epochs ($\mathsf{T}_{\mathsf{ep}}[w] \neq \perp$), the client puts a "dummy update" with an "old" search token $x := F(\mathsf{k}, w||\mathsf{T}_{\mathsf{ep}}[w])$ in val. We place val at addr derived using $\mathfrak{s}_e||1$ as the 1st slot in the list for w at epoch e.

If Srch() gets a "dummy update" ($(\mathsf{addr}, \mathsf{val}) \neq (\perp, \perp)$), but no actual update has happened yet ($\mathsf{EDB}[\mathsf{addr}] = \perp$), the servers sets $\mathsf{EDB}[\mathsf{addr}] := \mathsf{val}$. This is to ensure that no actual update would be accidentally overwritten. The server decrypts each tuple at $H_1(\mathfrak{s}_e||\mathsf{ct})$ of EDB by $H_2(\mathfrak{s}_e||\mathsf{ct})$ with ct increasing from 1 until no data can be found with $\mathfrak{s}_e$.

For the first update at each epoch ($\mathsf{ct} = 1$), the decrypted x is a search token $\mathfrak{s}'_e$ for prior updates on w. The server will set $\mathfrak{s}_e$ as $\mathfrak{s}'_e$ and repeat the search to get back all "historical" updates, provided the token is valid ($\mathfrak{s}_e \neq 0^\lambda$). The retrieved files are inserted into or removed from the result set R according to their operations. Eventually, the server returns R to the client.

Update. To update (op, w, f) at current epoch e, if keyword w has never been updated ($\mathsf{T}_{\mathsf{ct}}[w] = \perp$) or not been updated at e ($\mathsf{T}_{\mathsf{ep}}[w] \neq e$), it sets $\mathsf{T}_{\mathsf{ct}}[w] := 0$. $\mathsf{T}_{\mathsf{ct}}[w]$ is then incremented. If this is the first update on w at e ($\mathsf{T}_{\mathsf{ct}}[w] = 1$) and there exist prior (actual) updates on w at epoch $\mathsf{T}_{\mathsf{ep}}[w] \neq \perp$, the client puts prior search token $F(\mathsf{k}, w||\mathsf{T}_{\mathsf{ep}}[w])$ in x. It happens even when x has been implanted in a "dummy update" during search at e. Otherwise, $x := 0^\lambda$. $\mathsf{T}_{\mathsf{ep}}[w]$ is then set to e.

The client then stores x along with the update tuple. With $F(\mathsf{k}, w||e)||\mathsf{T}_{\mathsf{ct}}[w]$ as K, the update token $\mathfrak{u}$ is $(\mathsf{addr}, \mathsf{val})$, where addr is a pseudorandom location derived from K, and val is encrypting $(\mathsf{op}||f||x)$ under a key derived from K. The update token instructs the server to update $\mathsf{EDB}[\mathsf{addr}] := \mathsf{val}$.

Analysis. The size of both search and update tokens is $O(1)$. The update complexity is $O(1)$. The search complexity is linear in the total number of updates regarding the searched keyword, which is deemed asymptotically optimal [10,19,26].

For search, the server learns the search pattern and the update history of the keyword. Such search leakage $\mathcal{L}^{\mathsf{Srch}}(w)$ is necessary and common for efficient DSSE schemes [10,29].

For a search token, $\mathfrak{s}_e$ from PRF is pseudorandom. Whether to provide $(\mathsf{addr}, \mathsf{val})$ from H_1 and H_2 depends on whether the keyword of the token was updated at prior epochs but not the current one. Formally, for a search token of w at epoch e, $\mathcal{L}^{\mathsf{SrchTkn}}(w) = (\neg(\mathsf{UpHist}_{<e}(w) \stackrel{?}{=} \emptyset)) \wedge (\mathsf{UpHist}_e(w) \stackrel{?}{=} \emptyset)$. The leakage is subsumed by the entire $\mathsf{UpHist}(w)$. The token leaks nothing else before being revealed to the server and executed over the encrypted database, causing $\mathcal{L}^{\mathsf{Srch}}(w)$.

The search token for epoch e can search over data encrypted from the system initialization to e due to the preparation of prior search tokens. Namely, those tuples updated at e, due to the functionality of the temporal search, can be retrieved by the holder of the search token at the same epoch. However, any update after e remains private until the subsequent search as the server has no knowledge of the PRF secret key, *i.e.*, E-DSSE provides epoch-based forward privacy.

The security proof of E-DSSE can be found in Appendix B.

Theorem 3 *Assuming F is a pseudorandom function family,* E-DSSE *is $\mathcal{L}$-adaptively-secure with epoch-based forward privacy, where* $\mathcal{L}^{\mathsf{Stp}} = \emptyset$, $\mathcal{L}^{\mathsf{Srch}}(w) = (\mathsf{sp}(w), \mathsf{UpHist}(w))$, $\mathcal{L}^{\mathsf{SrchTkn}}(w) = (\neg(\mathsf{UpHist}_{<e}(w) \stackrel{?}{=} \emptyset)) \wedge (\mathsf{UpHist}_e(w) \stackrel{?}{=} \emptyset)$, $\mathcal{L}^{\mathsf{Updt}}(\mathsf{op}, w, f) = \begin{cases} \emptyset & \text{if } w \notin W_{\mathsf{Srch}}(e) \\ \mathcal{L}^{\mathsf{Srch}}(w) & \text{otherwise} \end{cases}$, *and e is the epoch when the update happens, in the random oracle model.*

6 Forward-Private HSE with $O(1)$-Size Tokens

With more versatile tools (E-DSSE and ICKAE), we build our forward-private hybrid searchable encryption (FP-HSE) with $O(1)$-size search tokens. We also advocate a new rebuild step for better performance and security at a small writer cost.

The constant size of search tokens is naturally inherited from our ICKAE, as the ICKAE secret key size is independent of the number of classes. Our FP-HSE does not need to run multiple copies of IBE or expensive pairing operations for each traversal step [39]. We set the reader free from the redundant key management of DSSE instances (whose number could be as many as that of the classes). With ICKAE, the reader is free from multiple calls of IBE.Ext() for searching.

FP-HSE assumes a loosely synchronized clock that provides the current epoch number. To retain the epoch-based forward privacy of E-DSSE while avoiding decrypting ever-growing ICKAE ciphertexts, we borrow ideas from a recent work [3] (that aims for breach resistance) to devise our rebuild algorithm, on top of our basic HSE syntax (Definition 4):

$\underline{(\mathsf{ETkn}_i, \mathsf{st}'_i) \leftarrow \mathsf{Rebuild}(\mathsf{st}_i, i, \mathsf{k}_i)}$ *executed by the writer takes as input a class $i \in [n]$ with its state st_i and secret key k_i. It outputs an encrypted token set ETkn_i to be stored in the server and a new state st'_i to be stored locally.*

Rebuild() requires writers to update their encrypted search tokens stored in the server at the start of a new epoch. It incurs no additional leakage as the tokens remain encrypted. The cost for our Rebuild() is quite reasonable for writers as it only depends on the number of the keywords they allow the reader to search. In contrast, previous Rebuild() [3] reconstructs the entire database, including encrypted indices and data (not for multiple writers but breach resistance).

6.1 Construction from E-DSSE and ICKAE

Figure 6 details our FP-HSE construction.

Setup. RSetup() of the reader runs ICKAE.Setup() and outputs a public/secret key pair $(\mathsf{pk}, \mathsf{msk})$ via ICKAE.KeyGen().

Each writer i individually executes WSetup() that calls E-DSSE.Setup(). Beyond an encrypted database EDB_i, an encrypted token set ETkn_i is set up. EDB_i and ETkn_i are initially empty. Same as G-HSE, the writer state contains a

$\mathsf{RSetup}(1^\lambda, n)$

$\mathsf{param} \leftarrow \mathsf{ICKAE.Setup}(1^\lambda, n)$
return $(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{ICKAE.KeyGen}()$

$\mathsf{WSetup}(1^\lambda, i)$

$(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i) \leftarrow \mathsf{E\text{-}DSSE.Setup}(1^\lambda)$
return $(\mathsf{k}_i, \mathsf{st}_i, \mathsf{EDB}_i, \mathsf{ETkn}_i := \emptyset)$

$\mathsf{Rebuild}(\mathsf{st}_i, i, \mathsf{k}_i)$

get current epoch e; **parse** $\mathsf{st}_i = (B_i, \ldots)$
$\mathsf{ETkn}_i := \emptyset$
foreach w s.t. $B_i[w] = 1$
  $(\mathfrak{s}_{\mathsf{sse}}, \mathsf{st}'_i) \leftarrow \mathsf{E\text{-}DSSE.SrchTkn}(\mathsf{st}_i, \mathsf{k}_i, w)$
  $c \leftarrow \mathsf{ICKAE.Enc}(\mathsf{pk}, i, w||e, \mathfrak{s}_{\mathsf{sse}})$
  $\mathsf{ETkn}_i := \mathsf{ETkn}_i \cup \{c\}$
return $(\mathsf{ETkn}_i, \mathsf{st}'_i)$

$\mathsf{UpdtTkn}(\mathsf{pk}, \mathsf{st}_i, i, \mathsf{k}_i, \mathsf{op}, w, f)$

get current epoch e; **parse** $\mathsf{st}_i = (B_i, \ldots)$
$(\mathfrak{u}_{\mathsf{sse}}, \mathsf{st}'_i) \leftarrow \mathsf{E\text{-}DSSE.UpdtTkn}(\mathsf{st}_i, \mathsf{k}_i, \mathsf{op}, w, f)$
if $B_i[w] = 0$
  $(\mathfrak{s}_{\mathsf{sse}}, \mathsf{st}''_i) \leftarrow \mathsf{E\text{-}DSSE.SrchTkn}(\mathsf{st}'_i, \mathsf{k}_i, w)$
  $c \leftarrow \mathsf{ICKAE.Enc}(\mathsf{pk}, i, w||e, \mathfrak{s}_{\mathsf{sse}})$
  $B_i[w] := 1$
else $c := \perp$
$\mathfrak{u} := (i, c, \mathfrak{u}_{\mathsf{sse}})$
return $(\mathfrak{u}, \mathsf{st}''_i)$

$\mathsf{Updt}(\mathfrak{u}, \mathsf{EDB}, \mathsf{ETkn})$

parse $\mathfrak{u} = (i, c, \mathfrak{u}_{\mathsf{sse}})$
if $c \neq \perp$ **then** $\mathsf{ETkn}'_i := \mathsf{ETkn}_i \cup \{c\}$
$\mathsf{EDB}'_i \leftarrow \mathsf{E\text{-}DSSE.Updt}(\mathfrak{u}_{\mathsf{sse}}, \mathsf{EDB}_i)$
return $(\mathsf{EDB}', \mathsf{ETkn}')$

$\mathsf{SrchTkn}(\mathsf{msk}, S, w)$

get current epoch e
return $\mathfrak{s} \leftarrow \mathsf{ICKAE.Ext}(\mathsf{msk}, S, w||e)$

$\mathsf{Srch}(\mathfrak{s}, S, \mathsf{EDB}, \mathsf{ETkn})$

parse $\mathsf{EDB} = \{\mathsf{EDB}_i\}_{i\in[n]}$
parse $\mathsf{ETkn} = \{\mathsf{ETkn}_i\}_{i\in[n]}$
$\mathcal{R} := \emptyset$
foreach $i \in S$
  foreach $c \in \{\mathsf{ETkn}_i\}$
    $\mathfrak{s}_{\mathsf{sse}} \leftarrow \mathsf{ICKAE.Dec}(\mathfrak{s}, S, i, c)$
    if $\mathfrak{s}_{\mathsf{sse}} \neq \perp$
      (R, EDB'_i)
        $\leftarrow \mathsf{E\text{-}DSSE.Srch}(\mathfrak{s}_{\mathsf{sse}}, \mathsf{EDB}_i)$
      $\mathcal{R} := \mathcal{R} \cup R$
return $(\mathcal{R}, \mathsf{EDB}', \mathsf{ETkn})$

Figure 6: FP-HSE: Forward-Private Hybrid Searchable Encryption from E-DSSE and ICKAE

$|\mathcal{W}|$-bit list[7] B_i for keyword space $\mathcal{W}$ to indicate whether a keyword has ever been updated by writer i. B_i can be inferred from the E-DSSE state, denoted by $\mathsf{st}_i = (B_i, \ldots)$ in Figure 6.

The server integrates EDB_i and ETkn_i from each writer i.

Search. At the current epoch e, to retrieve files from writers in set S containing keyword w, the reader extracts the HSE search token $\mathfrak{s}$ as an aggregated key derived from a single run of $\mathsf{ICKAE.Ext}()$ taking S as the target set and $w||e$ as ID. Due to our ICKAE scheme, for each keyword w, only one search token can be delegated in a given epoch e.

The server uses $\mathfrak{s}$ to decrypt $\{\mathsf{ETkn}_i\}_{i\in S}$ and parses the successfully decrypted result as E-DSSE search token $\mathfrak{s}_{\mathsf{sse}}$. With $\mathfrak{s}_{\mathsf{sse}}$, the server executes $\mathsf{E\text{-}DSSE.Srch}()$ over the corresponding EDB_i and inserts the result to $\mathcal{R}$. At the end, $\mathcal{R}$ is returned to the reader as the search result of w.

Update. To update (w, f), it first generates an E-DSSE update token $\mathfrak{u}_{\mathsf{sse}}$ for current epoch e. If $B_i[w] = 0$, the search token of w has not been generated; writer i generates an E-DSSE search token $\mathfrak{s}_{\mathsf{sse}}$ for w, ICKAE-encrypts it as c under class i and ID $w||e$, and sets $B_i[w]$ to 1. The HSE update token $\mathfrak{u}$, containing c and $\mathfrak{u}_{\mathsf{sse}}$, is sent to the server. The server stores c into ETkn_i and runs $\mathsf{E\text{-}DSSE.Updt}()$ over EDB_i with $\mathfrak{u}_{\mathsf{sse}}$.

Rebuild. At a new epoch, writer i initializes an empty ETkn_i. For every active keyword indicated by B_i, its E-DSSE search token is generated and encrypted by ICKAE with respect to class i, keyword w, and the current epoch e. The ICKAE ciphertexts are inserted into ETkn_i and then sent to the server. The server replaces the out-of-date encrypted token set of i.

Selective Rebuild. Not only can the reader confine the writer subset to search for, but also the writer can choose to confine the data that the reader can access at certain epochs, say, excluding draft data to be finalized at later epochs. This can be done by preparing encrypted E-DSSE search tokens only for those keywords that allow to be accessed at the next epoch during the rebuild, and skipping any generation of E-DSSE search tokens (and thus ICKAE encryption) during the update.

[7] This bit list is much smaller than, say, keeping copies in the "Sent" folder.

6.2 Analysis

Efficiency. An FP-HSE search traverses the target token sets, which is linear in the number of active keywords encrypted by the target classes, before executing an E-DSSE search, which is linear in the number of updates on the keyword. Thanks to the compactness of an ICKAE decryption key, the search token is constant-size, independent of the number of classes to be searched. For update, the token size and the time complexity are both constant. FP-HSE thus realizes sublinear (in the database size) search and update efficiency.

No interaction between the reader and any writer is ever needed. The reader only manages a single key pair.

Security. The setup only leaks the class identifiers $\{i\}_{i\in[n]}$.

For a corrupted writer (*i.e.*, $i \in I_c$), the update tuple is revealed. For any update from non-corrupted writers, $\mathsf{HSE.UpdtTkn}()$ runs $\mathsf{E\text{-}DSSE.UpdtTkn}()$, which generates an E-DSSE update token. Due to the epoch-based forward privacy of E-DSSE, the whole E-DSSE update process reveals *nothing* if the keyword has not been searched during the epoch. While $\mathsf{HSE.UpdtTkn}()$ might execute $\mathsf{E\text{-}DSSE.SrchTkn}()$, by our FP-HSE construction, $\mathsf{E\text{-}DSSE.SrchTkn}()$ is always executed after $\mathsf{E\text{-}DSSE.UpdtTkn}()$ at the same epoch. It means that the E-DSSE search token implants no dummy updates, so it incurs no extra leakage. The usage of the bit list B leaks whether the keyword has been updated by a class (*i.e.*, $\mathsf{UpHist}(i, w) \stackrel{?}{=} \emptyset$), which is unrelated to any prior searches.

FP-HSE thus retains the epoch-based forward privacy[8].

If an FP-HSE search on w has been issued over a subset S at epoch e, we consider $w \in W_{\mathsf{Srch}}(e)$ in E-DSSE for any updates on w at e from any $i \in S$. As G-HSE, the FP-HSE search over S leaks the E-DSSE search leakage (from target search tokens) and the searched keyword (from ICKAE decryption key).

The rebuild leaks nothing more than prior updates since 1) E-DSSE search tokens are all generated when there exist prior updates ($\mathsf{UpHist}_{<e}(w) \neq \emptyset$ in E-DSSE) and no update has happened at the new epoch ($\mathsf{UpHist}_{e}(w) = \emptyset$), covering the E-DSSE search-token-generation leakage; 2) the refreshed token set is ICKAE-encrypted; and 3) prior updates have already revealed the set of active keywords of a writer.

If writer i is corrupted, the (inevitable) leakage contains the historical updates of i (*i.e.*, $\mathsf{UpdtBy}(i)$). Theorem 4 formally asserts the security of FP-HSE with the proof in Appendix C.

Theorem 4 FP-HSE *is* $\mathcal{L}_{\mathsf{hse}}$*-adaptively-secure and forward private if* ICKAE *is* IND-CPA- *and* IND-ANON*-secure, and* E-DSSE *is* $\mathcal{L}_{\mathsf{sse}}$*-adaptively-secure with epoch-based forward privacy, where* $\mathcal{L}^{\mathsf{Stp}}_{\mathsf{hse}}(n) = \{i\}_{i\in[n]}$, $\mathcal{L}^{\mathsf{Corr}}_{\mathsf{hse}}(i) = \{\mathsf{UpdtBy}(i)\}$, $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{hse}}(w,S) = \{i, \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w), w\}_{i\in S}$, *and* $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{hse}}(i,\mathsf{op},w,f) =$
$$\begin{cases} \{i,\mathsf{op},w,f\} & \text{if } i \in I_c, \\ \{i, \mathsf{UpHist}(i,w) \stackrel{?}{=} \emptyset, \mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}(\mathsf{op},w,f)\} & \text{otherwise.} \end{cases}$$

It is possible to hide $\mathsf{UpHist}(i,w) \stackrel{?}{=} \emptyset$ if the writer follows the selective rebuild strategy, *i.e.*, no ICKAE ciphertext is generated during updates, and only E-DSSE search tokens for the searchable updates are encrypted during rebuilding.

Leakage-Abuse Attacks and Mitigation. Some early encrypted search solutions rely on property-preserving encryption, which suffer from statistical or ground-truth attacks. Afterward, most leakage-abuse attacks target SSE schemes for range queries (*e.g.*, see [38]), which is out of our scope.

Recent attacks [6, 32] do not require strong knowledge assumptions on plaintext data, but the volume pattern is essential for their accuracy. The modular design of HSE makes it compatible with the evolving volume-hiding DSSE [25, 37]. For example, the padding and batch-update approaches [37] work with our FP-HSE, as we could treat the updates of each epoch as a batch of updates. Meanwhile, leakage-suppression techniques [23] are proposed, which generically apply to DSSE.

7 Experiment and Deployment

We implement G-HSE and FP-HSE in Python. We instantiate DSSE and IBE for G-HSE with a dynamic variant of Π_{bas} [13] and Boneh–Franklin IBE [8], respectively. PRFs and keyed hash functions are instantiated with HMAC-SHA-256. We use PyCrypto 2.6.1 and Charm 0.50 library (charm-crypto.io) for cryptographic operations and MNT224 curve for pairings with a 96-bit security level. Our platform is Ubuntu 16.04 running over Intel Core i7-4790 3.60GHz CPU, 16GB RAM, and 1TB HDD. Results are averaged over 10 independent runs.

7.1 Performance over Real-World Datasets

Dataset. To echo our motivating applications of HSE, we start with evaluations over three real-world multi-writer datasets used by prior security and privacy research [14, 28, 31, 32].

Diabetes Dataset (EHR) [36]. 130 US hospitals contribute 101766 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. No time information was provided; we evolve the epoch every 1000 updates of records.

Room Climate Dataset (Sensor) [31]. It contains 540364 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. The epoch advances every hour.

Enron Email Dataset [1]. It consists of $5.1 \cdot 10^5$ emails sent by 146 employees. We let each employee/writer keep an active keyword space of the top 100 most frequent keywords (excluding stop words), resulting in a dataset with 3112196 keyword-email pairs. The epoch advances every month.

For comparison, we evaluate PKSE [2] and SPCHS [39] as baseline multi-writer solutions[9]. We only generate the searchable ciphertexts of keywords for them, as payload encryption is inexplicit in the original works. We stress again that SPCHS has neither forward privacy nor confined search. HSE, providing both, still shows its superiority in efficiency.

Build and Update. For the encrypted database generation, Figure 7 shows HSE surpasses SPCHS and PKSE by orders of magnitude. Concretely, FP-HSE is 23× (resp. 20×) for EHR, 98× (resp. 82×) for Sensor, and 75× (resp. 71×) for Enron, faster than SPCHS (resp. PKSE), yet a bit slower than G-HSE (see Section 7.2). It also reflects the update efficiency of HSE as we parse the procedure as updates on records.

Search. The reader can select arbitrary subsets of writers to search. Figure 8 evaluates the search performance with varying subset sizes. The search time of all schemes increases with the subset size since the sum of records, the number of active keywords, and the result size grow accordingly. FP-HSE outperforms SPCHS (resp. PKSE) by 3.6× (resp. 34×) for EHR, 18× (resp. 772×) for Sensor, and 2.4× (resp. 213×) for Enron. G-HSE is even faster because IBE takes one less pairing operation than ICKAE for decryption.

[8] For (level-II) backward privacy [11] , one can generally follow the two-roundtrip approach used by prior DSSE schemes [11, 14]: the writer updates with $\mathsf{PKE.Enc}(\mathsf{op}||f)||x$ instead of $(\mathsf{op}||f||x)$, with PKE being set up by the reader. The reader could remove deleted files locally after decrypting.

[9] We omit FP-PKSE [40] as it is even slower than PKSE [2] by orders of magnitude. We follow the new-ibe-2-peks transform [2] to construct PKSE, which simultaneously encrypts random messages for consistency.

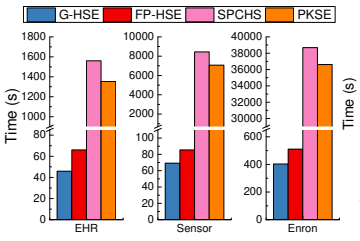


Figure 7: Building Time

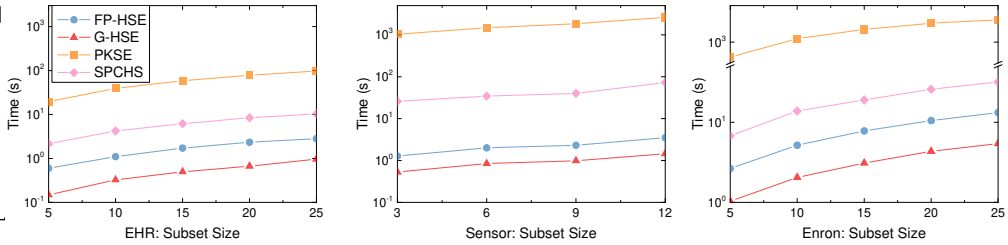
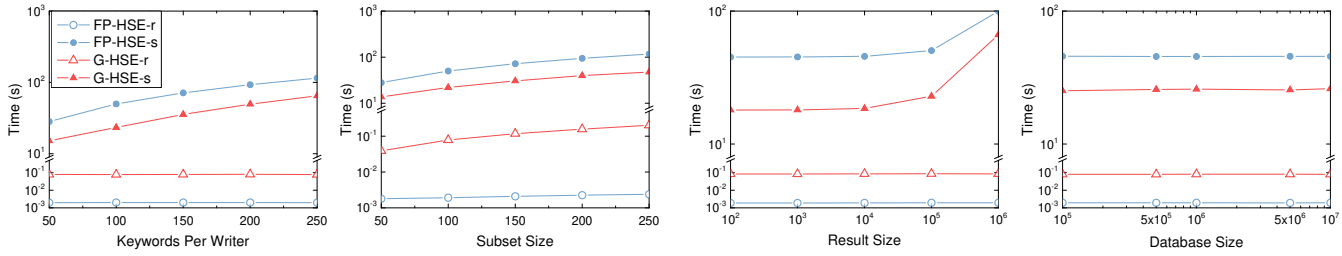
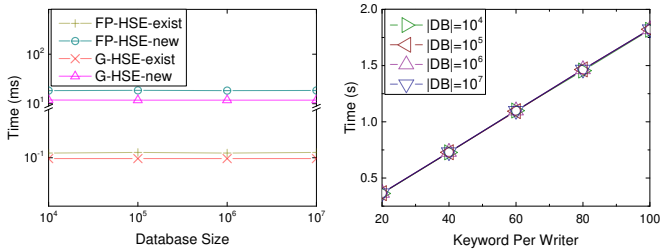


Figure 8: Real-World Search Performance of Multi-Writer SE



(a) (b) (c) (d)

Figure 9: Search Performance of HSE over Synthetic Datasets



(a) Update (b) Rebuild

Figure 10: Update and Rebuild Time of HSE

Evidently, HSE is more suitable for real-world VLDB applications where the data size per keyword is huge.

Other parameters, *i.e.*, the result size and the number of active keywords, are not necessarily correlated (*e.g.*, a large result size of a keyword does not mean more active keywords). We will study their influences on HSE with synthetic datasets.

Rebuild. To rebuild, FP-HSE only pays ∼395ms for EHR, ∼505ms for Sensor, and ∼1.79s for Enron per writer, depending on their active keywords. The cost of enjoying confined search and forward privacy in FP-HSE is quite affordable.

7.2 Performance over Synthetic Datasets

Parameters of Interest. We consider how sizes of the active keyword set per writer $|K|$, target writer subset $|S|$, search results $|R|$, and the database $|\mathsf{DB}|$ influence HSE's performance.

Dataset. Our synthetic databases contain $10^4 - 10^7$ records. The keyword space is set to one-hundredth of the database size. Unless otherwise specified, we define a set of classes with a fixed size $n = 500$, namely, 500 writers in the scheme, and assume each writer contributes an equal number of random records to the database, including those to be searched. Especially for FP-HSE, the random records are inserted into the database at randomly chosen epochs from 1 to 1000.

Search. We measure the search time of G-HSE and FP-HSE. Figure 9 splits the time into *reader-side* (G-HSE/FP-HSE-r) and *server-side* (G-HSE/FP-HSE-s) for a clear illustration. In practice, the server-side searching can be parallelizable as the encrypted token sets and the databases can be categorized according to the classes/writers. The search time will be shorter with multi-thread techniques. Our evaluation solely reflects the performance with no industrial optimizations.

Number of Active Keywords Per Writer. Figure 9a shows how search time changes with the number of active keywords per writer. We let each writer encrypt a varying number $|K| \in \{50, 100, 150, 200, 250\}$ of keywords, with fixed $|S| = 100$, $|R| = 10^5$, and $|\mathsf{DB}| = 10^7$. For both schemes, the server takes more time for a larger $|K|$ due to the larger encrypted token sets, while reader overheads are independent of $|K|$.

Subset Size. Figure 9b shows the search time for different subset sizes $|S| \in \{50, 100, 150, 200, 250\}$ of writers, with fixed $|K| = 100$, $|R| = 10^5$, and $|\mathsf{DB}| = 10^7$. The reader-side search time increases with $|S|$ for traversing more encrypted tokens. It is lower in FP-HSE than G-HSE, which reflects the time saved in $\mathsf{SrchTkn}()$ for extracting an aggregate ICKAE key for subset S (in FP-HSE) versus $|S|$ IBE decryption keys (in G-HSE). Precisely, $\mathsf{ICKAE.Ext}()$ needs two exponentiations and $|S|$ multiplications, while $|S|$ IBE key extraction [8] takes $|S|$ exponentiations. Figures 9a, 9c, and 9d also show this reduction. ICKAE makes HSE efficient for the reader.

Result Size. Figure 9c shows the search time for different result sizes $|R| \in \{10^2, 10^3, 10^4, 10^5, 10^6\}$, with fixed $|K| = 100$, $|S| = 100$, and $|\mathsf{DB}| = 10^7$. For both schemes, the server-side search overheads are dominated by IBE/ICKAE decryptions until the result size becomes extremely large (over 10^5). The reader-side overhead is independent of $|R|$.

Database Size. Figure 9d confirms that the search time is independent of the database sizes $|\mathsf{DB}| \in \{10^5, 5\cdot 10^5, 10^6, 5\cdot 10^6, 10^7\}$, fixing $|S| = 100$, $|K| = 100$, and $|R| = 10^3$.

Update and Rebuild. Figure 10a shows the time of a single update for databases with $\{10^4, 10^5, 10^6, 10^7\}$ records and a fixed class size $n = 500$. We consider two cases: 1) the keyword is new for the class (G-HSE/FP-HSE-new); 2) the search token of the keyword exists (G-HSE/FP-HSE-exist). This is to confirm the advantage that $\mathsf{UpdtTkn}()$ of both G-HSE and FP-HSE do not repeatedly encrypt the same search token.

The update overheads for both (FP-HSE/G-HSE) are independent of the database. "G-HSE-exist" and "FP-HSE-exist" reflect the update time of underlying DSSE schemes, which are almost the same. Considering IBE and ICKAE encryption for new search tokens, "FP-HSE-new" is only 6ms slower than "G-HSE-new." Such a tiny difference is acceptable to the individual writer and necessary to benefit other aspects of FP-HSE, particularly, the compact token and the reduction in search time for the reader, as we emphasized above. The baseline is, they are both in the order of milliseconds (ms).

Lastly, we consider the rebuild procedure of FP-HSE, which contributes to saving search time on top of forward privacy. Setting $n = 100$, we measure the rebuild time per writer with a varying number of active keywords per writer $|K| \in \{20, 40, 60, 80, 100\}$ over databases of size $|\mathsf{DB}| \in \{10^4, 10^5, 10^6, 10^7\}$. As Figure 10b shows, the rebuild time per writer is linear in the number of his/her active keywords, independent of the database size (from overlapping curves).

Concluding Remarks. For the reader, the search of FP-HSE is faster than G-HSE at a tiny cost in the update time of writers. The rebuild time for FP-HSE depends on the number of keywords of individual writers and is acceptable in practice. In short, our evaluations show that both G-HSE and FP-HSE provide a favorable level of search and update efficiency.

7.3 Applications and Deployment

The efficiency of HSE, as shown by our experiments (*e.g.*, over Enron), makes it applicable to scenarios valuing timeliness, *e.g.*, secure messaging/email. To search over (encrypted) messages from different contacts, current practices require a client to download or store all messages. HSE improves it with quick searches confined to only writers of interest.

HSE benefits data-sharing over sensitive information contributed by different parties. It generalizes to novel applications, *e.g.*, collective intelligence and crowdsourcing, where a reader makes decisions upon data from different sources in a secure way. In short, HSE extends the original vision of structured encryption [15] for supporting controlled disclosure.

We discuss HSE deployment for health records, which desires encrypted databases. We consider a hierarchical structure for this scenario: each hospital builds a local HSE database with clinical data from its affiliated doctors, while the hospital authority sets up a global one with records contributed by multiple hospitals (after pretreatments). Each doctor/hospital independently writes to the local/global database via HSE updates. A search interface provided by the hospital/authority enables doctors/researchers to read records from the local/global database for better diagnoses/contact-tracing analyses. With forward privacy and confined search, no (sensitive) information beyond what is intended to share will be revealed.

8 Final Remarks and Future Research

We formulate hybrid searchable encryption (HSE) for the sublinear search complexity in symmetric searchable encryption (SSE) and the support of multiple writers in public-key searchable encryption (PKSE). Utilizing the two new building blocks E-DSSE and ICKAE, our HSE instantiation features epoch-based forward privacy and constant-size search tokens.

Compared with PKSE that traverses the whole encrypted database, we reduce the number of costly IBE/ICKAE decrypt operations to the number of distinct keywords encrypted by the writers targeted by a search. Such dependency is arguably intrinsic. When keywords form the basis for the keyword search, without any writer pre-establishing (unique) secret material with the reader, the multiple writers need to public-key encrypt independently, differentiating their ciphertexts from others for the confined search requirement. We see our work making searchable encryption more deployable and as a foundation for extensions to more functionalities.

As a new paradigm, there are many interesting questions to explore. One might explore how to conduct secure deduplication (*e.g.*, via message-locked encryption [42]) across HSE databases with entries contributed by different writers, *e.g.*, when a patient sought medical advice from different hospitals.

For key rotation in the worry of key compromise, we can generalize our epoch-based definition to capture the epoch evolution it triggers. To our knowledge, no prior study considers efficient key rotation for searchable encryption.

For foundational issues, it would be interesting to formulate an epoch-based version of backward privacy, which might inspire new directions in the ongoing research of backward privacy. For example, one may investigate non-interactive and efficient techniques for different flavors of backward privacy.

In the broader context of cryptography, we did not explore hierarchical ICKAE due to space limitation, which would be useful in various applications [4, 7, 17, 18], particularly epoch-based revocation. Another direction is to devise ICKAE in the standard model without the restriction on key extraction query and/or equipped with both function privacy [9] and anonymous-ciphertext indistinguishability [16].

We focus on showcasing a new approach to multi-writer searchable encryption. With our E-DSSE, the next to investigate is devising an ICKAE scheme "compatible" with the SPCHS paradigm, further boosting the efficiency of HSE.

Finally, it is interesting to explore alternative formulations of multi-writer searchable encryption with sublinear search.

References

[1] Enron dataset. https://www.cs.cmu.edu/~enron.

[2] Michel Abdalla, Mihir Bellare, Dario Catalano, Eike Kiltz, Tadayoshi Kohno, Tanja Lange, John Malone-Lee, Gregory Neven, Pascal Paillier, and Haixia Shi. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. *J. Cryptology*, 21(3), 2008.

[3] Ghous Amjad, Seny Kamara, and Tarik Moataz. Breach-resistant structured encryption. *PoPETs*, 2019(1):245–265, 2019.

[4] Nuttapong Attrapadung, Jun Furukawa, and Hideki Imai. Forward-secure and searchable broadcast encryption with short ciphertexts and private keys. In *AsiaCrypt*, pages 161–177, 2006.

[5] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *CRYPTO*, pages 535–552, 2007.

[6] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *NDSS*, 2020.

[7] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *EuroCrypt*, pages 506–522, 2004.

[8] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. *SIAM J. Comput.*, 32(3):586–615, 2003.

[9] Dan Boneh, Ananth Raghunathan, and Gil Segev. Function-private identity-based encryption: Hiding the function in functional encryption. In *CRYPTO Part II*, pages 461–478, 2013.

[10] Raphael Bost. $\sum o\varphi o\varsigma$: Forward secure searchable encryption. In *CCS*, pages 1143–1154, 2016.

[11] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*, pages 1465–1482, 2017.

[12] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In *EuroCrypt*, pages 255–271, 2003.

[13] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS*, 2014.

[14] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *CCS*, pages 1038–1055, 2018.

[15] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *AsiaCrypt*, pages 577–594, 2010.

[16] Sherman S. M. Chow. Removing escrow from identity-based encryption. In *PKC*, pages 256–276, 2009.

[17] Sherman S. M. Chow, Volker Roth, and Eleanor Gilbert Rieffel. General certificateless encryption and timed-release encryption. In *SCN*, pages 126–143, 2008.

[18] 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 Trans. Para. Dis. Sys.*, 25(2):468–477, 2014.

[19] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS*, pages 79–88, 2006.

[20] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[21] Alex Escala, Gottfried Herold, Eike Kiltz, Carla Ràfols, and Jorge Luis Villar. An algebraic framework for Diffie-Hellman assumptions. *J. Cryptology*, 30(1), 2017.

[22] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *USENIX Security*, 2021.

[23] Marilyn George, Seny Kamara, and Tarik Moataz. Structured encryption and dynamic leakage suppression. In *EuroCrypt Part III*, pages 370–396, 2021.

[24] Ariel Hamlin, abhi shelat, Mor Weiss, and Daniel Wichs. Multi-key searchable encryption, revisited. In *PKC Part I*, pages 95–124, 2018.

[25] Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In *EuroCrypt Part II*, pages 183–213, 2019.

[26] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *CCS*, pages 965–976, 2012.

[27] Jonathan Katz and Nan Wang. Efficiency improvements for signature schemes with tight security reductions. In *CCS*, pages 155–164, 2003.

[28] Miran Kim, Junghye Lee, Lucila Ohno-Machado, and Xiaoqian Jiang. Secure and differentially private logistic regression for horizontally distributed data. *IEEE Trans. Inf. Forensics Secur.*, 15:695–710, 2020.

[29] Russell W. F. Lai and Sherman S. M. Chow. Forward-secure searchable encryption on labeled bipartite graphs. In *ACNS*, pages 478–497, 2017.

[30] Antonis Michalas. The lord of the shares: combining attribute-based encryption and searchable encryption for flexible data sharing. In *SAC*, pages 146–155, 2019.

[31] Philipp Morgner, Christian Müller, Matthias Ring, Björn Eskofier, Christian Riess, Frederik Armknecht, and Zinaida Benenson. Privacy implications of room climate data. In *ESORICS Part II*, pages 324–343, 2017.

[32] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security*, pages 127–142, 2021.

[33] Zhiwei Shang, Simon Oya, Andreas Peter, and Florian Kerschbaum. Obfuscated access and search patterns in searchable encryption. In *NDSS*, 2021.

[34] Dawn Xiaodong Song, David A. Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE S&P*, pages 44–55, 2000.

[35] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, 2014.

[36] Beata Strack, Jonathan P. DeShazo, Chris Gennings, Juan L. Olmo, Sebastian Ventura, Krzysztof J. Cios, and John N. Clore. Impact of HbA1c measurement on hospital readmission rates: analysis of 70,000 clinical database patient records. *BioMed Research Intl'*, 2014.

[37] Jiafan Wang and Sherman S. M. Chow. Simple storage-saving structure for volume-hiding encrypted multi-maps. In *DBSec*, pages 63–83, 2021.

[38] Jiafan Wang and Sherman S. M. Chow. Forward and backward-secure range-searchable symmetric encryption. *PoPETs*, 2022(1), 2022. To appear.

[39] 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 Trans. Information Forensics and Security*, 10(9):1993–2006, 2015.

[40] Ming Zeng, HaiFeng Qian, Jie Chen, and Kai Zhang. Forward secure public key encryption with keyword search for outsourced cloud storage. *IEEE Trans. Cloud Comp.*, 2019. Early Access.

[41] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security*, pages 707–720, 2016.

[42] Yongjun Zhao and Sherman S. M. Chow. Updatable block-level message-locked encryption. *IEEE Trans. Dependable Secur. Comput.*, 18(4):1620–1631, 2021.

A Security Proof of ICKAE (Theorem 2)

Given an IND-CPA adversary $\mathcal{A}$, we build $\mathcal{B}$ below that solves the n-bilinear Diffie-Hellman exponent (n-BDHE) problem, *i.e.*, computes $[r\alpha^{n+1}]_t$ from $([r]_2, \{[\alpha^i]_1, [\alpha^i]_2\}_{i\in[2n]\setminus\{n+1\}})$.

1. Sample $\hat{i} \leftarrow\$ [n]$, $s, \zeta \leftarrow\$ \mathbb{Z}_p$. Initialize an empty map T_H.
2. Set $[\delta]_2 := s[\alpha]_2$ (*i.e.*, $\delta = s\alpha$) and $[\gamma]_2 := [\zeta]_2 - [\alpha^{\hat{i}}]_2$.
3. Set $\mathsf{pk} := ([\gamma]_2, [\delta]_2)$. Compute $[\alpha^{n+1}]_t = [\alpha]_1[\alpha^n]_2$.
4. Set $\mathsf{param} := (\{[\alpha^i]_1, [\alpha^i]_2\}_{i\in[2n]\setminus\{n+1\}}, [\alpha^{n+1}]_t)$.
5. Simulate the oracle G by lazy programming, *i.e.*, on a new query $[u]_t$, return $G([u]_t) := v$ where $v \leftarrow\$ \{0,1\}^\lambda$.
6. Upon H query on (id_k, b):
   (a) If $((\mathsf{id}_k, b), [h]_1, x, d, \theta)$ exists in T_H, return $[h]_1$.
   (b) Else if $((\mathsf{id}_k, \bar{b}), \cdot, \cdot, d', \cdot)$ exists in T_H, set $d = \bar{d}'$.
   (c) If id_k never exists in T_H, flip a fair coin that outputs a bit $d = 1$ with probability $1/2$, $d = 0$ otherwise.
   (d) Finally, pick $x \leftarrow\$ \mathbb{Z}_p$.
      i. If $d = 1$, flip a coin that outputs a bit $\theta = 1$ with probability ρ, 0 otherwise. If $\theta = 1$, return $[h]_1 := (1/s)[\alpha^n + x]_1$. If $\theta = 0$, return $[x]_1$.
      ii. If $d = 0$, set $\theta = \perp$, return $[h]_1 := [x]_1$.
   Record $((\mathsf{id}_k, b), [h]_1, x, d, \theta)$ in the map T_H.
7. Upon $\mathsf{Ext}O$ query on $S \subseteq [n]$ and id_k, first issue H queries for $(\mathsf{id}_k, 0)$ and $(\mathsf{id}_k, 1)$ to ensure that they exist in T_H.
   (a) If $\hat{i} \notin S$, get $((\mathsf{id}_k, b), [h]_1, x, 0, \theta)$ from T_H, return $\mathsf{ak} := \zeta\sum_{j\in S}[\alpha^{n+1-j}]_1 - \sum_{j\in S}[\alpha^{n+1+\hat{i}-j}]_1 + sx[\alpha]_1$ and b. Since $\hat{i} \notin S$, ak can be computed from param.
   (b) If $\hat{i} \in S$, retrieve $((\mathsf{id}_k, b), [h]_1, x, 1, \theta)$ from T_H. If $\theta = 1$, $[h]_1 = (1/s)[\alpha^n + x]_1$, return $\mathsf{ak} := \zeta\sum_{j\in S}[\alpha^{n+1-j}]_1 - \sum_{j\in S\setminus\{\hat{i}\}}[\alpha^{n+1+\hat{i}-j}]_1 + x[\alpha]_1$, which is $\zeta\sum_{j\in S}[\alpha^{n+1-j}]_1 - \sum_{j\in S}[\alpha^{n+1+\hat{i}-j}]_1 + [\alpha^{n+1}]_1 + x[\alpha]_1 = \gamma\sum_{j\in S}[\alpha^{n+1-j}]_1 + \delta[h]_1$, and b. If $\theta = 0$, abort and output a random $\mathbb{G}_t$ element.
8. Receive $(\mathsf{st}, m_0, m_1, i^*, \mathsf{id}^*) \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{pk})$.
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 from T_H $((\mathsf{id}^*, 0), [h_0^*]_1, x_0^*, d_0^*, \theta_0)$ and $(\mathsf{id}^*, 1), [h_1^*]_1, x_1^*, d_1^*, \theta_1)$.
11. If $(d_{b^*}^* = 1) \wedge (\theta_{b^*} = 1)$ for some $b^* \in \{0,1\}$, abort.
12. Simulate the ciphertext as follows.

(a) Sample $b \leftarrow\$ \{0,1\}$. Set $[c_1]_2 := [r]_2$, $[c_2]_2 := \zeta[r]_2$. ($[r(\gamma+\alpha^{i^*})]_2 = [r(\zeta-\alpha^{i^*}+\alpha^{i^*})]_2 = [r\zeta]_2$.)
(b) Set $c_{3,d} := m_b \oplus v_d^*$ with $v_d^* \leftarrow\$ \{0,1\}^\lambda, \forall d \in \{0,1\}$.
(c) Return $c^* := ([c_1]_2, [c_2]_2, c_{3,0}, c_{3,1})$.

13. Receive $b' \leftarrow \mathcal{A}^{\mathsf{Ext}O}(\mathsf{st}, c^*)$.
14. Randomly pick one query $[u^*]_t$ to the G oracle.
15. Output $[u^*]_t + sx^*_{b'}[\alpha]_1[r]_2$.

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

Now it remains to analyze the probability that $\mathcal{B}$ does not abort. For an $\mathsf{Ext}O$ query (S, id_k), no matter whether $\hat{i} \in S$ or not, there is at most one kind of random-oracle assignment $H(\mathsf{id}_k, b)$ depending on a bit d that does not lead to abortion. Firstly, d is chosen by a fair coin independent of b. By construction, only the key for a particular bit b will be returned. Due to the restriction that each id_k can only appear once as part of an $\mathsf{Ext}O$ query, if such a query does not abort, the other abortion case for the same query would never happen.

An $\mathsf{Ext}O$ query (S, id_k) only aborts when $\hat{i} \in S$ and $((\mathsf{id}_k, b), [h]_1, x, 1, \theta = 1)$ exists in T_H, which happens with probability at most $(1-\rho)$. $\mathcal{B}$ can simulate the challenge ciphertext when $(\hat{i} = i^*) \wedge (\theta_{b^*} = 0)$ for b^* s.t. $d^*_{b^*} = 1$, which happens with probability $(1-\rho)/n$. When there are at most q_{E} $\mathsf{Ext}O$ queries, $\mathcal{B}$ simulates faithfully with probability $\rho^{q_{\mathsf{E}}}(1-\rho)/n$, which is maximized at $\rho = 1 - 1/(q_{\mathsf{E}}+1)$, leading to the probability of at least $1/ne(1+q_{\mathsf{E}})$ for large q_{E}.

There is an alternative simulation with probability $\rho = 1$ (*i.e.*, oracle H always picks $\theta = 1$) and the rule of "If $(d^*_{b^*} = 1) \wedge (\theta_{b^*} = 1)$ for some $b^* \in \{0,1\}$, abort." removed. In the original simulation, both $c_{3,d}$ terms for $d \in \{0,1\}$ in the challenge ciphertext involve hashing the n-BDHE solution with $G()$. With the changed simulation, while all key extraction queries can be answered, one of the $c_{3,d}$ terms does not involve the n-BDHE solution. However, when the internal variable $d^*_{b^*}$ corresponding to either oracle H's responses $[h^*_0]_1$ or $[h^*_0]_1$ is information-theoretically hidden, the chance for $\mathcal{A}$ to figure out b via asking the "bad" query to oracle G is at most $1/2$.

B Security Proof of E-DSSE (Theorem 3)

We derive a hybrid sequence from the real game $\mathbf{Real}_{\mathcal{A}}(1^\lambda)$ to the ideal hybrid $\mathbf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^\lambda)$. By showing that each game (except the first) is indistinguishable from its previous one, we conclude that the adversary cannot distinguish $\mathbf{Real}_{\mathcal{A}}(1^\lambda)$ from $\mathbf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^\lambda)$ with non-negligible probability.

We assume that the adversary $\mathcal{A}$ makes at most q_1 and q_2 queries to H_1 oracle and H_2 oracle, respectively. The output length of H_1, H_2, and PRF F are μ_1, μ_2, and λ, respectively. We denote $F(\mathsf{k}, w||e)$ by $\mathsf{k}_{w,e}$ for ease of interpretation.

$\mathbf{Hybrid}_0$: As **Real**, $\Pr[\mathbf{Hybrid}_0 = 1] = \Pr[\mathbf{Real}_{\mathcal{A}}(1^\lambda) = 1]$.

$\mathbf{Hybrid}_1$: Instead of invoking F with k to generate $\mathsf{k}_{w,e}$, $\mathbf{Hybrid}_1$ randomly picks a λ-bit string when given a new query of $w||e$ and stores it in a map $\mathsf{T}_{\mathsf{w,e}}$ to answer the same query next time. If $\mathcal{A}$ can distinguish $\mathbf{Hybrid}_0$ from $\mathbf{Hybrid}_1$, we can build a reduction to distinguish between a pseudorandom function and a truly random function.

$\mathbf{Hybrid}_2$: Instead of querying $H_1(\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w])$ for addr, $\mathbf{Hybrid}_2$ picks a random μ_1-bit string and stores it in $\mathsf{T}_{\mathsf{addr}}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ if $w \notin W_{\mathsf{Srch}}(e)$. $\mathsf{T}_{\mathsf{addr}}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ will be returned whenever $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ is queried for this case.

Otherwise, w has been searched at epoch e, and $\mathsf{k}_{w,e}$ has been revealed. $\mathbf{Hybrid}_2$ checks whether $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ exists in table $\mathbf{H_1}$ for the random oracle H_1 and programs $\mathbf{H_1}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ with a random μ_1-bit string if it is not. $\mathbf{H_1}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ will be returned for this case.

For the first search of the keyword at an epoch, program $\mathbf{H_1}[\mathfrak{s}_e||\mathsf{ct}] := \mathsf{T}_{\mathsf{addr}}[\mathfrak{s}_e||\mathsf{ct}]$ for $\mathsf{ct} = 1, 2, \ldots$, until $\mathsf{T}_{\mathsf{addr}}[\mathfrak{s}_e||\mathsf{ct}] = \perp$. The search token $\mathfrak{s}_e$ is generated in the same way as $\mathsf{k}_{w,e}$ during the update. The procedure repeats for all unprogrammed entries regarding $\mathfrak{s}_e$ updated previously.

For the case when w has been searched at epoch e, since all subsequent entries related to $\mathsf{k}_{w,e}$ can be programmed immediately, $\mathcal{A}$ observes no inconsistency. For the other case, addr for $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ is generated, but $\mathbf{H}_1$ will not be programmed until the next search of w. If $\mathcal{A}$ queries $\mathbf{H}_1$ for $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ before such searches, the returned value, as a randomly picked μ_1-bit string, will have an overwhelming probability to be different from the one programmed for $\mathbf{H}_1[\mathsf{k}_{w,e}||\mathsf{ct}]$ (*i.e.*, $\mathbf{H}_1[\mathfrak{s}_e||\mathsf{ct}]$) later during the search. As $\mathbf{Hybrid}_2$ is the same as $\mathbf{Hybrid}_1$ except when the above inconsistency (denoted by BAD) is observed, $\Pr[\mathbf{Hybrid}_1 = 1] - \Pr[\mathbf{Hybrid}_2 = 1] \leq \Pr[\mathsf{BAD}]$.

Since $\mathfrak{s}_e$ / $\mathsf{k}_{w,e}$ is a random λ-bit string, the probability that the adversary queries $\mathbf{H}_1$ for it equals $2^{-\lambda}$. Since $\mathcal{A}$ makes at most q_1 queries to the H_1 oracle, we have $\Pr[\mathsf{BAD}] \leq q_1/2^\lambda$, *i.e.*, $\mathbf{Hybrid}_1$ and $\mathbf{Hybrid}_2$ are indistinguishable.

$\mathbf{Hybrid}_3$: Instead of querying H_2 at $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ for computing $\mathsf{val} := (\mathsf{op}||f||x) \oplus H_2(\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w])$, $\mathbf{Hybrid}_3$ picks a μ_2-bit string and stores it in $\mathsf{T}_{\mathsf{val}}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$, if $w \notin W_{\mathsf{Srch}}(e)$. $\mathsf{T}_{\mathsf{val}}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ will be returned for this case.

Otherwise, w has been searched at epoch e and $\mathsf{k}_{w,e}$ has been revealed. $\mathbf{Hybrid}_3$ checks whether $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ exists in table $\mathbf{H_2}$ for H_2 and programs $\mathbf{H_2}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ with a random μ_2-bit string if it is not. Return $\mathbf{H_2}[\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]]$ for this case.

For the first search of the keyword at an epoch, $\mathbf{H_2}$ is programmed by setting $\mathbf{H_2}[\mathfrak{s}_e||\mathsf{ct}] := \mathsf{T}_{\mathsf{val}}[\mathfrak{s}_e||\mathsf{ct}]$ for $\mathsf{ct} = 1, 2, \ldots$, until $\mathsf{T}_{\mathsf{val}}[\mathfrak{s}_e||\mathsf{ct}] = \perp$. $\mathfrak{s}_e$ is generated in the same way as $\mathsf{k}_{w,e}$ during the update. The procedure repeats for all unprogrammed entries regarding $\mathfrak{s}_e$ updated previously.

Likewise, the probability that $\mathcal{A}$ discovers the inconsistency of the random oracle H_2 is at most $q_2/2^\lambda$, which is negligible. $\mathbf{Hybrid}_2$ and $\mathbf{Hybrid}_3$ are thus indistinguishable.

$\mathbf{Hybrid}_4$: It is exactly **Ideal**, *i.e.*, the simulator $\mathcal{S}$ generates a view only based on $\mathcal{L}$. $\mathcal{L}^{\mathsf{SrchTkn}}$ is a bit indicating whether

the keyword was updated at prior epochs but not the current one. $\mathcal{L}^{\mathsf{Srch}}$ contains search pattern sp and update history UpHist. $\mathcal{L}^{\mathsf{Updt}}$ is the same as $\mathcal{L}^{\mathsf{Srch}}$ if the keyword has been searched at the current epoch; otherwise, it leaks nothing.

Different from $\mathbf{Hybrid}_3$, $\mathfrak{s}_e$ / $\mathsf{k}_{w,e}$ is randomly sampled during search token generation. Meanwhile, $\mathcal{S}$ decides whether to sample and include $(\mathsf{addr}, \mathsf{val})$ in $\mathfrak{s}$ based on $\mathcal{L}^{\mathsf{SrchTkn}}$. Instead of mapping $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ to entries of $\mathsf{T}_{\mathsf{addr}}$ and $\mathsf{T}_{\mathsf{val}}$, $\mathbf{Hybrid}_4$ maps the global timestamp when $\mathsf{k}_{w,e}||\mathsf{T}_{\mathsf{ct}}[w]$ is updated to entries of two timestamp-indexed dictionaries, storing random strings generated for addr and val respectively. During the search, entries in random oracles regarding $\mathsf{k}_{w,e}$ is programmed as per its timestamps, $\mathsf{sp}(w)$, and $\mathsf{UpHist}(w)$.

$\mathbf{Hybrid}_3$ and $\mathbf{Hybrid}_4$ are indistinguishable since both hybrids 1) for update, output two random strings (for addr and val); and 2) for search, output $\mathfrak{s}_e$ with the same distributions, while $(\mathsf{addr}, \mathsf{val})$ (if provided in the search token) consists of random strings programmed by the random oracle. Thus, $\Pr[\mathbf{Hybrid}_3 = 1] = \Pr[\mathbf{Hybrid}_4 = 1]$.

By combining the above (in)equalities, we have $|\Pr[\mathbf{Real}_{\mathcal{A}}(1^\lambda) = 1] - \Pr[\mathbf{Ideal}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda)$.

C Security Proof of FP-HSE (Theorem 4)

Let $\mathcal{L}_{\mathsf{sse},i} = \{\mathcal{L}^{\mathsf{Stp}}_{\mathsf{sse},i}, \mathcal{L}^{\mathsf{SrchTkn}}_{\mathsf{sse},i}, \mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}, \mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}\}$ and $\mathcal{S}_{\mathsf{sse},i}$ be leakage functions and the simulator of the i-th E-DSSE instance for $i \in [n]$, respectively. The initially empty EDB_i and ETkn_i leak nothing. $\mathcal{L}^{\mathsf{Stp}}_{\mathsf{hse}}(n)$ only leaks the set of class identifiers.

With $W_{\mathsf{Srch}}(i,e) = \{w | (\mathsf{Srch}, w, S, e) \in \mathcal{H} \wedge i \in S\}$ as the set of keywords that has been searched at epoch e for writing subset containing i, we parse the E-DSSE update leakage as $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}(\mathsf{op}, w, f) = \emptyset$, if $w \notin W_{\mathsf{Srch}}(i,e)$; $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w)$, otherwise.

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

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

For $k \in \{0, \ldots, q-1\}$, Hist_{k+1} can fall into three cases:

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

(2)$\mathsf{Srch}\mathcal{O}_b$ on $(\{S_j, w_j, e\}_{j\in\{0,1\}})$: As the target classes of any search leaks, it will only be answered when $S = S_0 = S_1$.

As the keywords to be searched are revealed (due to the delegation of ICKAE decryption key, similar to the case for IBE), the oracle only answers the query when w_0 and w_1 are identical, *i.e.*, $w = w_0 = w_1$. 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 E-DSSE search regarding E-DSSE search tokens of w encrypted with ICKAE during previous $\mathsf{Updt}\mathcal{O}$ or rebuild. The challenger returns the ICKAE decryption key of $(S, w||e)$. $\mathbf{Hybrid}_k = \mathbf{Hybrid}_{k+1}$, since the views of $\mathcal{A}$ are identical.

(3) $\mathsf{Updt}\mathcal{O}_b$ on $(\{i_j, \mathsf{op}_j, w_j, f_j, e\}_{j\in\{0,1\}})$: The oracle answers the queries when $i = i_0 = i_1$, as the class will be leaked during update. Obviously, if i has been corrupted, $\mathcal{A}$ will have the knowledge of the update tuples, and the oracle only answers when two tuples are identical in this case.

Assume i has not been corrupted until timestamp k. When only one of the keywords has been searched at current epoch e (*i.e.*, contained in $W_{\mathsf{Srch}}(i,e)$), it returns $\perp$ as it results in different $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}$ (recall the E-DSSE update leakage). Similarly, the oracle returns $\perp$ when only one of the keywords has been updated before for leading to different $\mathsf{UpHist}(i,w) \stackrel{?}{=} \emptyset$.

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

- 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 $w_0 = w_1$. The oracle invokes the E-DSSE simulator $\mathcal{S}_{\mathsf{sse},i}$ with $\mathcal{L}^{\mathsf{Srch}}_{\mathsf{sse},i}(w_b)$ to simulate $\mathfrak{u}_{\mathsf{sse}}$.
- Otherwise, the oracle simply calls the E-DSSE simulator $\mathcal{S}_{\mathsf{sse},i}$ with $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{sse},i}(\mathsf{op}, w_b, f) = \emptyset$ to simulate $\mathfrak{u}_{\mathsf{sse}}$.

In the above two cases, the oracle processes $\mathfrak{s}_{\mathsf{sse}}$ differently depending on whether both keywords have been updated: If $\mathsf{UpHist}(i,w_0) = \mathsf{UpHist}(i,w_1) = \emptyset$, the oracle calls the E-DSSE simulator $\mathcal{S}_{\mathsf{sse},i}$ to simulate i's E-DSSE search token with $\mathcal{L}^{\mathsf{SrchTkn}}_{\mathsf{sse},i}(w_b) = 0$. The token is ICKAE-encrypted as c, treating $w_b||e$ as the identity; Otherwise, $c := \perp$.

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

For $\mathsf{Epoch}\mathcal{O}$ (not explicitly included in $\mathcal{H}$), if any writer refuses to rebuild at any epoch, nothing will be changed for $\mathcal{A}$. The rebuild essentially ICKAE-encrypts new search tokens for keywords updated before. As it is invoked at (the start of) a new epoch during which no keyword has been updated, the E-DSSE simulator outputs search tokens with $\mathcal{L}^{\mathsf{SrchTkn}}_{\mathsf{sse},i}(w_b) = 1$. $\mathcal{A}$ cannot gain any non-negligible advantage from such a step due to IND-CPA and IND-ANON security of ICKAE.

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

Forward privacy of HSE (Definition 8) constrains the update leakage when updating keywords that have not been searched at the same epoch. In FP-HSE, for this case (*i.e.*, $w \notin W_{\mathsf{Srch}}(i,e)$), the leakage $\mathcal{L}^{\mathsf{Updt}}_{\mathsf{hse}}$ is $\{i, \mathsf{UpHist}(i,w) \stackrel{?}{=} \emptyset\}$, fulfilling Definition 8. Thus, FP-HSE is forward-private.