# Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation

David Cash[1], Joseph Jaeger[1], Stanislaw Jarecki[2], Charanjit Jutla[3], Hugo Krawczyk[3], Marcel-Cătălin Roşu[3], and Michael Steiner[3]

[1]Rutgers University
[2]University of California Irvine
[3]IBM Research

October 17, 2014

**Abstract**

We design and implement dynamic symmetric searchable encryption schemes that efficiently and privately search server-held encrypted databases with tens of billions of record-keyword pairs. Our basic theoretical construction supports single-keyword searches and offers asymptotically optimal server index size, fully parallel searching, and minimal leakage. Our implementation effort brought to the fore several factors ignored by earlier coarse-grained theoretical performance analyses, including low-level space utilization, I/O parallelism and goodput. We accordingly introduce several optimizations to our theoretically optimal construction that model the prototype's characteristics designed to overcome these factors. All of our schemes and optimizations are proven secure and the information leaked to the untrusted server is precisely quantified. We evaluate the performance of our prototype using two very large datasets: a synthesized census database with 100 million records and hundreds of keywords per record and a multi-million webpage collection that includes Wikipedia as a subset. Moreover, we report on an implementation that uses the dynamic SSE schemes developed here as the basis for supporting recent SSE advances, including complex search queries (e.g., Boolean queries) and richer operational settings (e.g., query delegation), in the above terabyte-scale databases.

## 1 Introduction

Background. Searchable symmetric encryption (SSE) allows one to store data at an untrusted server and later search the data for records (or documents) matching a given keyword while maintaining privacy. Many recent works [4–6,8,10,15,16,18,20,22] studied SSE and provided solutions with varying trade-offs between security, efficiency, and the ability to securely update the data after it has been encrypted and uploaded. These constructions aim at practical efficiency, in contrast to generic cryptographic tools like homomorphic encryption or multiparty computation which are highly secure but not likely to be efficient in practice soon.

Large data sizes motivate storage outsourcing, so to be useful an SSE scheme must scale well. Existing SSE schemes employ only symmetric cryptography operations and standard data structures and thus show potential for practical efficiency, but obstacles remain. While most constructions have theoretically optimal search times that scale only with the number of documents matching the query, the performance of their implementations on large datasets is less clear. Factors like I/O latency, storage utilization, and the variance of real-world dataset distributions degrade the practical performance of theoretically efficient SSE schemes. One critical source of inefficiency in practice (often ignored in theory) is a complete lack of locality and parallelism: To execute a search, most prior SSE schemes *sequentially* read each result from storage at a

最重要的是局部性的完
全缺失和并行

pseudorandom position, and the only known way to avoid this *while maintaining privacy* involves padding the server index to a prohibitively large size.

Contributions. We give the first SSE implementation that can encrypt and search on datasets with tens of billions of record/keyword pairs. To design our scheme, we start with a new, simple, theoretical SSE construction that uses a generic dictionary structure to already achieve an asymptotic improvement over prior SSE schemes, giving optimal leakage, server size, search computation, and parallelism in search. This starting point can be seen as a generalization and simplification of the more ad-hoc techniques of [4]. We show how to make the scheme *dynamic*, meaning that the data can be changed after encryption: Our scheme can easily support additions to the data, as well as deletions via revocation lists.

Because the scheme uses a generic dictionary that itself has no security properties, it allows for several extensions and modifications with only small changes to the security proofs. In particular, our implementation effort showed that disk I/O utilization remained a bottleneck which prevented scaling; so we extend our basic construction to improve locality and throughput. These extensions preserve privacy with slightly different leakages that we analyze with formal security proofs. Below we describe the techniques behind results in more detail, starting with the new theoretical scheme that we extend later, and then compare our results to prior work.

Basic construction. Our scheme is very simple (see Figure 5): It associates with each record/keyword pair a pseudorandom label, and then for each pair stores the encrypted record identifier with that label in a generic dictionary data structure. We derive the labels so that the client, on input a keyword to query, can compute a keyword-specific short key allowing the server to search by first recomputing the labels, then retrieving the encrypted identifiers from the dictionary, and finally decrypting the matching encrypted record identifiers. The only information leaked to the server by the encrypted index (other than the indexes of records matching a query) is the number of items in the dictionary, i.e. the number of record/keyword pairs in the data. This scheme is easy to implement correctly (and with parallel searching) because we make no security demands on the dictionary thus allowing instantiations as applications demand.

Extensions for external storage. To compute the results of a keyword search with $r$ matches, our basic scheme requires $r$ retrievals from the dictionary for pseudorandom labels. Assuming $O(1)$ cost of a dictionary retrieval, this is asymptotically optimal. However, in implementations this may still be slow when the dictionary is stored in external memory (i.e., a block device like a HDD), because each random-looking retrieval will generate a disk read. This is in contrast to a plaintext system which could store all of the matches in a single contiguous area of memory. 为什么明文就可以呢

In view of this reality we extend our scheme to use external storage more carefully while maintaining privacy. We first show how to securely "pack" related results together via a padding strategy to reduce the number of dictionary retrievals.

We found that even this modification was too slow for the datasets we targeted, and in particular we noticed that real data-sets exhibit extreme variability in the number of matches for a keyword: There were typically many keywords matching very few documents, then some keywords matching a significant fraction of the entire database. Our padding strategy becomes unsatisfactory because the (many) keywords matching only a few results create a lot of padding, and the searches that return a large number of results still trigger a large number of dictionary retrievals. 导致填充效率不是很好

To address this we introduce further modifications that replace dictionary reads with array reads when processing large numbers of results. These modifications result in a slightly different, but intuitively acceptable (and perhaps even better) leakage profile that we discuss below.

Extension for updates. We observe that our scheme easily extends to allow for additions to the data after it has been uploaded. We only have to arrange that the client can compute the labels for the new data to be added, which it sends to the server for to be added to the dictionary. This requires either client state or communication proportional to the total number of keywords ever added or deleted. To support deletions we maintain a (pseudorandom) revocation list at the server that allows the server to filter out results that should be deleted. To actually reclaim space we re-encrypt the entire database periodically.

SSE方案对比

| Scheme | Security | Ind Leak | Dyn.? | Dyn Leak | Index Size | Search Time/Comm | Dyn. Comm |
|---|---|---|---|---|---|---|---|
| CGKO'06-1 [8] | NonAd | $m, N$ | No | — | $O(N+m)$ | $O(r), O(1)$ | — |
| CGKO'06-2 [8] | Ad | $Mn$ | No | — | $O(Mn)$ | $O(r), O(r)$ | — |
| CK'10 [6] | Ad | $m, n, M$ | No | — | $O(Mn)$ | $O(r), O(r)$ | — |
| LSDHJ'10 [22] | Ad | $m, n$ | Yes | no proof | $O(mn)$ | $O(m), O(1)$ | $O(\lvert \mathsf{W}_{\mathsf{id}} \rvert)$ |
| KO'12 [18] | Ad(UC) | $n, M$ | No | — | $O(Mn)$ | $O(r), O(1)$ | — |
| KPR'12 [16] | $\mathrm{Ad}^{\mathrm{ro}}$ | $m, N$ | Yes | $\mathrm{EP}(\mathsf{W}_{\mathsf{id}})$ | $O(N+m)$ | $O(r), O(1)$ | $O(\lvert \mathsf{W}_{\mathsf{id}} \rvert)$ |
| KP'13 [15] | $\mathrm{Ad}^{\mathrm{ro}}$ | $m, n$ | Yes | minimal | $O(mn)$ | $O((r \log n)/p), O(1)$ | $O(\lvert \mathsf{W}_{\mathsf{id}} \rvert + m \log n)$ |
| Basic ($\Pi_{\mathrm{bas}}$ here) | NonAd, $\mathrm{Ad}^{\mathrm{ro}}$ | $N$ | No | — | $O(N)$ | $O(r/p), O(1)$ | — |
| Basic Adp ($\Pi^{\mathrm{ro}}_{\mathrm{bas}}$ here) | Ad | $N$ | No | — | $O(N)$ | $O(r/p), O(r)$ | — |
| Basic Dyn ($\Pi^{\mathrm{dyn}}_{\mathrm{bas}}, \Pi^{\mathrm{dyn,ro}}_{\mathrm{bas}}$ here) | NonAd, $\mathrm{Ad}^{\mathrm{ro}}$ | $N$ | Yes | minimal | $O(N)$ | $O((r+d_w)/p), O(1)$ | $O(\lvert \mathsf{W}_{\mathsf{id}} \rvert + m \log n)$ |

Figure 1: Comparison of some SSE schemes. Many leakages can be replaced by upper bounds and some search times assume interaction when the original paper was non-interactive. Legend: In security, "Ad" means adaptive security, $\mathrm{Ad}^{\mathrm{ro}}$ means adaptive security in the random oracle model, and NonAd means non-adaptive security. Ind Leakage is leakage from encrypted database only. Search comm. is the size of the message sent from client ($O(r)$ from the server is inherent.) ro means random oracle model, $n$ = # documents, $N = \sum_w \lvert \mathsf{DB}(w) \rvert$, $m = \lvert \mathsf{W} \rvert$, $M = \max_w \lvert \mathsf{DB}(w) \rvert$, $r = \lvert \mathsf{DB}(w) \rvert$ for the query $w$, $p$ = # processors, $\lvert \mathsf{W}_{\mathsf{id}} \rvert$ = # keyword changes in an update, $\mathrm{EP}(\mathsf{W}_{\mathsf{id}})$ = structural equality pattern of changed keywords (see discussion at the end of Section 4), $d_w$ = the number of times the searched-for keyword has been added/deleted.

Other applications. Recent constructions of SSE supporting more complex queries [4] and multi-client settings [14] use SSE as a black-box. Thus our data structures and associated operations (including support for dynamic databases) are readily available to support terabyte-scale databases in these much richer/complex encrypted-search settings (see end of Section 2).

Implementation. Our implementation remains efficient on *two orders of magnitude larger* datasets than the most scalable previous work [4], resulting in the first implementation of SSE on terabyte-scale databases containing tens of billions of indexed record/keyword pairs. We report on our prototype design and experimental results in Section 5.

Comparison to prior work. In Figure 1 we compare our basic theoretical scheme to prior work. The basic scheme $\Pi_{\mathrm{bas}}$ generalizes and greatly simplifies an approach implicit in [4], which complicated the analysis by demanding security properties of the underlying data structures.

For a database with $N$ record/keyword pairs, our basic scheme $\Pi_{\mathrm{bas}}$ produces an encrypted index of optimal size $O(N)$, leaks only the size $N$ and the matching record id's, and processes a search with $r$ results in optimal $O(r)$ time, assuming $O(1)$-cost for dictionary retrievals. Searching is trivial to parallelize with any number of processors.

Most prior schemes leak additional information, like the number of unique keywords, the size of the largest number of matches for a keyword, and so on. Some of these works also pad their encrypted indexes to be (worst-case) quadratic in their input size, which is totally impractical for large datasets. A notable issue with most prior work was a difficulty with parallelism: Other than [4], parallel searching was only achieved by two works that needed quadratic padding. Works like [8] required walking through an encrypted linked list and were not parallelizable at all. See the "Ind Leak", "Index Size", and "Search Time" columns in Figure 1.

The only prior dynamic schemes either had an impractically large index [15] or leaked the *structure* of the added documents [16], meaning that the server learned, say, the pattern of which keywords appear in which documents as they are added, which is a severe form of leakage compared to the usual SSE leakage of facts like the total database size. Our dynamic extension maintains the optimal index size and only leaks basic size information (and not document structure, as in [16]). Unlike prior dynamic schemes, ours does not reclaim space after each deletion - rather, we envision applications where deletions are relatively rare or, more generally, where a periodic complete re-encryption of the data is performed (re-encryption may be desirable to mitigate the leakage from updates with any dynamic SSE scheme).

```
Fn(j, x)    // PRFReal_F(λ)
  $K \leftarrow T[j]$ ; $y \leftarrow F(K, x)$
  ret $y$

Fn(j, x)    // PRFRand_F(λ)
  $y \leftarrow R[j, x]$
  ret $y$
```

```
E(j, m),    // RCPAReal_Π(λ)
  $K \leftarrow T[j]$ ; $C \leftarrow\$ \mathsf{Enc}(K, m)$
  ret $C$

E(j, m),    // RCPARand_Π(λ)
  $C \leftarrow\$ \{0,1\}^{\ell(\lambda, |m|)}$
  ret $C$
```

Figure 2: Oracles for games $\mathrm{PRFReal}_F, \mathrm{PRFRand}_F, \mathrm{RCPAReal}_\Pi$, and $\mathrm{RCPARand}_\Pi$.

More on related work. The notion of SSE we consider has its origins in work by Song, Wagner, and Perrig [20]. Several schemes since have improved upon the security and efficiency offered by the original schemes. The most similar to our construction is that of Chase and Kamara [6], and Cash et al [4]. Chase and Kamara also uses a dictionary, but in a different way and with an impractical level of padding for large datasets. Cash et al implements a scheme similar to our basic construction, but do not address updates nor, as we show in Section 5.5, does their approach achieve the same level of practical scalability.

There is also a related line of work on searchable public-key encryption starting with [3], all of which do not scale due to linear-time searching. The version of SSE we deal with inherently leaks the identifiers of documents that match a query, as well as when a query is repeated. It is possible to hide even this information using private information retrieval [7] or oblivious RAM [11]. Several recent improvements to oblivious RAM move it further towards practicality [12,21], but it is far from clear that they are competitive with SSE schemes at scale as one must implement the plaintext searching as an abstract RAM program and then run this program underneath the ORAM scheme without leaking information, say, via timing.

Organization. Preliminary definitions are given in Section 2. Our non-dynamic (i.e., static) SSE constructions are given in Section 3, and the dynamic extensions are given in Section 4. Finally we report on our implementation in Section 5.

## 2 Definitions and Tools

The security parameter is denoted $\lambda$. We will use the standard notions of variable-input-length PRFs and symmetric encryption schemes (c.f. [17]). For these primitives we make the simplifying assumption that their keys are always in $\{0,1\}^\lambda$, and that key generation for the encryption scheme picks a random key. By *efficient* we mean probabilistic poly-time in $\lambda$. We write $\mathrm{negl}(\lambda)$ for a negligible function in $\lambda$. Some of our constructions will be analyzed in the random oracle model [1], with the random oracle denoted $H$.

Games. We formalize some of our security notions using code-based games [2]. A game $G$ is defined by a collection of oracle procedures. A game is executed with an adversary (i.e., a randomized polynomial time algorithm) $A$. Running a game with adversary $A$ means running $A$ with access to the prescribed set of oracles, possibly with some restrictions on its access (for instance, if there is an **Init** oracle, we will always restrict the adversary to querying it once and before any other query). When $A$ halts and $G$ runs the **Final** procedure with the output $A$, and the output of the game is the output of **Final**, denote as $G^A$. When **Final** is omitted, we mean that it forwards its input as output (i.e., the adversary's output is the output of the game). When $G$ is a game we use the shorthand $Pr[G]$ for the probability that $G$ outputs 1.

In defining games we adopt the following notation: Boolean flags are initialized false. When $T$ is a dictionary, $T[j]$ denotes the data item associated with $j$, if there is one. When we write $x \leftarrow T[j]$ we mean that $x$ is assigned the value $T[j]$ if it exists, and otherwise a uniformly random value from the appropriate range is selected, stored at $T[j]$, and assigned to $x$.

PRFs and encryption. Our constructions will use as components pseudorandom functions and symmetric encryption schemes. For the former we formalize a variant of the usual definition using games

```
Init(DB)   // $\mathrm{SSECor}_\Pi(\lambda)$
  $(K, \mathsf{EDB}) \leftarrow\$ \mathsf{Setup}(\mathsf{DB})$
  ret EDB

Final()   // $\mathrm{SSECor}_\Pi(\lambda)$
  ret win
```

```
Up(op, in)   // $\mathrm{SSECor}_\Pi(\lambda)$
  $(\mathsf{EDB}; \tau) \leftarrow\$ \mathsf{Update}(K, \mathsf{op}, \mathsf{in}; \mathsf{EDB})$
  $\mathsf{DB} \leftarrow \mathsf{Apply}(\mathsf{DB}, \mathsf{op}, \mathsf{in})$
  ret $\tau$
```

```
Srch(w),   // $\mathrm{SSECor}_\Pi(\lambda)$
  $(V; \tau) \leftarrow\$ \mathsf{Search}(K, w; \mathsf{EDB})$
  if $V \neq \mathsf{DB}(w)$ then win $\leftarrow$ true
  ret $\tau$
```

Figure 3: Oracles for game $\mathrm{SSECor}_\Pi$.

PRFReal, PRFRand from Figure 2. We only consider algorithms $F$ that on input $K \in \{0,1\}^\lambda$,$x \in \{0,1\}^*$, output a string in $\{0,1\}^\lambda$. In the games the adversary chooses an index $j$ and an input $x \in \{0,1\}^*$ when querying **Fn**. In PRFReal the key stored at $T[j]$ is used to produce the output (being selected at random if necessary, as per our convention). In PRFRand responses are returned from entries in a dictionary, each initialized to a random value as it is used. Restricting this game to only allow queries with $j = 1$ collapses this to the usual definition, and our version is clearly asymptotically equivalent by a standard hybrid argument.

**Definition 1** *An algorithm $F$ is a* variable-input-length pseudorandom function *if for all efficient $A$, the function*

$$\mathbf{Adv}^{\mathrm{prf}}_{F,A}(\lambda) = \Pr[\mathrm{PRFReal}^A_F(\lambda) = 1] - \Pr[\mathrm{PRFRand}^A_F(\lambda) = 1]$$

*is negligible.*

An *encryption scheme* $\Pi$ (symmetric key) is a pair of efficient algorithms $(\mathsf{Enc}, \mathsf{Dec})$ such that for all $K \in \{0,1\}^\lambda$ and $m \in \{0,1\}^*$, $\mathsf{Dec}(K, \mathsf{Enc}(K,m)) = m$ with probability 1. We assume that $\Pi$ is *$\ell$-length regular*, meaning there exists a function $\ell$ such that $|\mathsf{Enc}(K,m)| = \ell(\lambda, |m|)$ for all $\lambda, m$, where $|\cdot|$ denotes string length.

We will assume access to an encryption scheme that has *pseudorandom ciphertexts under chosen-plaintext attack*, which we call *RCPA security.* As with PRFs, we formalize this in a multi-key variant that simplifies our reductions. This is defined using games $\mathrm{RCPAReal}_\Pi(\lambda)$ and $\mathrm{RCPARand}_\Pi(\lambda)$. In the first game, the adversary can query its **E** oracle with an index $j$ and message $m \in \{0,1\}^*$, and the game generates a ciphertext using a key from $T[j]$ (generated at random as necessary) and message $m$. In the second game, a fresh random string of the appropriate length $\ell(\lambda, m)$ is always returned, where $\Pi$ is $\ell$-regular.

**Definition 2** *An encryption scheme $\Pi = (\mathsf{Enc}, \mathsf{Dec})$ has* pseudorandom ciphertexts under chosen-plaintext attack *if for all efficient $A$, the function*

$$\mathbf{Adv}^{\mathrm{ind-rcpa}}_{\Pi,A}(\lambda) = \Pr[\mathrm{RCPAReal}^A_\Pi(\lambda) = 1] - \Pr[\mathrm{RCPARand}^A_\Pi(\lambda) = 1]$$

*is negligible.*

Our constructions can be proved secure assuming a type of key anonymity property, but RCPA is simpler and is anyway achieved by many efficient constructions like counter-mode.

SSE SCHEMES. We follow the formalization of Curtmola et al. [8] with some modifications discussed below. Below when an algorithm takes a set as input, we assume that it is represented by writing its elements in lexicographic order. A *database* $\mathsf{DB} = (\mathsf{id}_i, \mathsf{W}_i)_{i=1}^d$ is a $d$-tuple of identifier/keyword-set pairs where $\mathsf{id}_i \in \{0,1\}^\lambda$ and $\mathsf{W}_i \subseteq \{0,1\}^*$. When the $\mathsf{DB}$ under consideration is clear, we will write $\mathsf{W} = \bigcup_{i=1}^d \mathsf{W}_i$. For a keyword $w \in \mathsf{W}$, we write $\mathsf{DB}(w)$ for $\{\mathsf{id}_i : w \in \mathsf{W}_i\}$. We will always use $m = |\mathsf{W}|$ and $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$ to mean the number of keywords and the total number of keyword/document matches in $\mathsf{DB}$.

A *dynamic searchable symmetric encryption (SSE) scheme* $\Pi$ consists of an algorithm $\mathsf{Setup}$ and protocols $\mathsf{Search}$ and $\mathsf{Update}$ between the client and server fitting the syntax below. A *static* SSE scheme is exactly the same, but with no $\mathsf{Update}$ protocol. We assume that the server is deterministic, and that the client may hold some state between queries. Formally, the protocols are defined by two efficient next-message algorithms for the parties, but we will avoid making this explicit as the details in security definitions are straightforward when considering adversaries that follow the protocol.

$\mathsf{Setup}$ takes as input a database $\mathsf{DB}$, and outputs a secret key $K$ along with an encrypted database $\mathsf{EDB}$. In the search protocol the client takes as input the secret key $K$ and a query $w \in \{0,1\}^*$ and the server takes as input $\mathsf{EDB}$ and the server outputs a set of identifiers and the client has no output. In the $\mathsf{Update}$ protocol the client takes as input a key $K$, an operation $\mathsf{op} \in \{\mathsf{add}, \mathsf{del}, \mathsf{edit}^+, \mathsf{edit}^-\}$, a file identifier $\mathsf{id}$, and a set $\mathsf{W}_{\mathsf{id}}$ of keywords. These inputs represent the actions of adding a new file with identifier $\mathsf{id}$ containing keywords $\mathsf{W}_{\mathsf{id}}$, deleting the file with identifier $\mathsf{id}$, or add/removing the keywords in $\mathsf{W}_{\mathsf{id}}$ from an existing file. At the end of the $\mathsf{Update}$, the server outputs an updated encrypted database, and the client has no output.

We will write

$$(V;\tau) \leftarrow\!\$\, \mathsf{Search}(K, w; \mathsf{EDB})$$

to mean that $V$ and $\tau$ are sampled by running the search protocol with client input $K, w$ and server input $\mathsf{EDB}$ and letting $V$ be the server output and $\tau$ be the messages sent by the client. (We omit the messages sent by the server from $\tau$ since they can be calculated from $\tau$ and $\mathsf{EDB}$ when the server is deterministic.) Similarly we write

$$(\mathsf{EDB}';\tau) \leftarrow\!\$\, \mathsf{Update}(K, \mathsf{op}, \mathsf{in}; \mathsf{EDB})$$

to mean executing the update protocol with client inputs $K, \mathsf{op}, \mathsf{in}$ and server input $\mathsf{EDB}$, and then letting $\mathsf{EDB}'$ be the server output and $\tau$ be the messages from the client.

We say that an SSE scheme is *correct* if the search protocol returns the (current) correct results for the keyword being searched (i.e., $\mathsf{DB}(w)$), except with negligible probability. We formalize this with game $\mathrm{SSECor}(\lambda)$ in Figure 3, where the adversary is additionally restricted to never add a duplicate identifier, add redundant keyword to an existing identifier, delete with currently nonexistent identifier, or delete a keyword from an identifier that does not match it. In this game we use the function $\mathsf{Apply}$, which takes as input a $\mathsf{DB}$, an operation $\mathsf{op}$ of one of the allowed values, and inputs $\mathsf{in}$ for that operation. It outputs an updated version of $\mathsf{DB}$ with the operation applied to it.

**Definition 3** *An SSE scheme* $\Pi$ *is* correct *if for all efficient* $A$

$$\mathbf{Adv}^{\mathrm{sse-cor}}_{\Pi,A}(\lambda) = \Pr[\mathrm{SSECor}^{A}_{\Pi}(\lambda) = 1]$$

*is a negligible function.*

Discussion. For simplicity our formalization of SSE does not model the storage of the actual document payloads. The SSE literature varies on its treatment of this issue, but in all cases one can augment the schemes to store the documents with no additional leakage beyond the number and length of the payloads. Compared to others we model also modifications of documents ($\mathsf{edit}^+, \mathsf{edit}^-$) in addition to add and delete of complete documents ($\mathsf{add}, \mathsf{del}$) as this can lead to more efficient protocols with reduced leakage.

The correctness definition for SSE requires the server to learn the $\mathsf{id}$s of the results. One could define correctness to require the *client* to learn the $\mathsf{id}$s instead. The two approaches are essentially equivalent assuming that encrypted documents are of fixed length.

Security. Security [6, 8, 16] follows the real/ideal simulation paradigm and is parametrized by a *leakage function* $\mathcal{L}$ that describes what a secure protocol is allowed to leak. Formally, $\mathcal{L}$ is an algorithm used in the simulation game below. The definition uses two games, $\mathrm{SSEReal}_{\Pi}(\lambda)$ and $\mathrm{SSESim}_{\mathcal{L},S}(\lambda)$. In the first game, the adversary chooses a $\mathsf{DB}$ input and recieves $\mathsf{EDB}$, and then can adaptively query **Up** and **Srch**, which return transcripts from the $\mathsf{Update}$ and $\mathsf{Search}$ protocols respectively (recall that here a "transcript" means client messages only as the server message are efficiently computable from them and $\mathsf{EDB}$). The **Final** oracle halts the computation and returns the bit $b$ chosen by the adversary. In this game we place the same correctness restrictions on the adversary to prevent it from issuing invalid dynamic operations.

**Definition 4** *A dynamic SSE scheme* $\Pi$ *is* $\mathcal{L}$-secure against adaptive attacks *if for all efficient* $A$ *there exists an efficient* $S$ *such that*

$$\mathbf{Adv}^{\mathrm{sse-adap}}_{\Pi,A,S}(\lambda) = \Pr[\mathrm{SSEReal}^{A}_{\Pi}(\lambda) = 1] - \Pr[\mathrm{SSESim}^{A}_{\mathcal{L},S}(\lambda) = 1]$$

*is a negligible function. For a static SSE scheme* $\Pi$ *the definition is the same, except* $A$ *is only not allowed to query* **Up** *in the games.*

```
$\underline{\mathbf{Init}(\mathsf{DB})}$   // $\mathrm{SSEReal}_\Pi(\lambda)$
$(K, \mathsf{EDB}) \leftarrow\$ \mathsf{Setup}(\mathsf{DB})$
$\mathbf{ret}\ \mathsf{EDB}$

$\underline{\mathbf{Up}(\mathsf{op}, \mathsf{in})}$   // $\mathrm{SSEReal}_\Pi(\lambda)$
$(\mathsf{EDB}; \tau) \leftarrow\$ \mathsf{Update}(K, \mathsf{op}, \mathsf{in}; \mathsf{EDB})$
$\mathbf{ret}\ \tau$

$\underline{\mathbf{Srch}(w),}$   // $\mathrm{SSEReal}_\Pi(\lambda)$
$(V; \tau) \leftarrow\$ \mathsf{Search}(K, w; \mathsf{EDB})$
$\mathbf{ret}\ \tau$

$\underline{\mathbf{Final}(b)}$   // $\mathrm{SSEReal}_\Pi(\lambda)$
$\mathbf{output}\ b$
```

```
$\underline{\mathbf{Init}(\mathsf{DB})}$   // $\mathrm{SSESim}_{\mathcal{L},S}(\lambda)$
$(\alpha, \mathsf{st}_\ell) \leftarrow\$ \mathcal{L}(\mathsf{DB})$ ; $(\mathsf{EDB}, \mathsf{st}_s) \leftarrow\$ S(\alpha)$
$\mathbf{ret}\ \mathsf{EDB}$

$\underline{\mathbf{Up}(\mathsf{op}, \mathsf{in})}$   // $\mathrm{SSESim}_{\mathcal{L},S}(\lambda)$
$(\alpha, \mathsf{st}_\ell) \leftarrow\$ \mathcal{L}(\mathsf{st}_\ell, \mathsf{op}, \mathsf{in})$ ; $(\tau, \mathsf{st}_s) \leftarrow\$ S(\mathsf{st}_s, \alpha)$
$\mathbf{ret}\ \tau$

$\underline{\mathbf{Srch}(w),}$   // $\mathrm{SSESim}_{\mathcal{L},S}(\lambda)$
$(\alpha, \mathsf{st}_\ell) \leftarrow\$ \mathcal{L}(\mathsf{st}_\ell, w)$ ; $(\tau, \mathsf{st}_s) \leftarrow\$ S(\mathsf{st}_s, \alpha)$
$\mathbf{ret}\ \tau$

$\underline{\mathbf{Final}(b)}$   // $\mathrm{SSESim}_{\mathcal{L},S}(\lambda)$
$\mathbf{output}\ b$
```

Figure 4: Oracles for games $\mathrm{SSEReal}_\Pi(\lambda), \mathrm{SSESim}_{\mathcal{L},S}(\lambda)$.

We define *$\mathcal{L}$-security against non-adaptive attacks* in the same way, except that in both games $A$ must choose all of its queries at the start, $\mathcal{L}$ takes them all as input, and $S$ uses the output of $\mathcal{L}$ to generate $\mathsf{EDB}$ and the transcripts at the same time. Formally, in the game $\mathrm{SSENAReal}_\Pi$, $A$ gets to make a single query to **Init** consisting of a $\mathsf{DB}$ and a list of $q$ queries, where each query is of the form $(\mathsf{op}, \mathsf{in})$ or $(\mathsf{srch}, w)$, with the latter representing search queries. The game sets $(K, \mathsf{EDB}) \leftarrow \mathsf{Setup}(\mathsf{DB})$. Then it processes each query, computing $\tau_i$ by running either $\mathsf{Update}$ or $\mathsf{Search}$ and then overwriting $\mathsf{EDB}$ if the query was for $\mathsf{Update}$. It returns $(\mathsf{EDB}, (\tau_1, \ldots, \tau_Q))$ to $A$, which can query a bit to **Final** that becomes out the output of the game.

In the corresponding simulation game $\mathrm{SSENASim}_{\mathcal{L},S}(\lambda)$, $A$ issues the same single query to **Init** which is answered by computing $\alpha \leftarrow\$ \mathcal{L}(\mathsf{DB}, \mathbf{q})$ and returning the output of $S(\alpha)$. The adversary queries its **Final** oracle once as before.

**Definition 5** *A dynamic SSE scheme* $\Pi$ *is* $\mathcal{L}$-secure against nonadaptive attacks *if for all efficient $A$ there exists an efficient $S$ such that*

$$\mathbf{Adv}^{\mathrm{sse-adap}}_{\Pi,A,S}(\lambda) = \Pr[\mathrm{SSENAReal}^A_\Pi(\lambda) = 1] - \Pr[\mathrm{SSENASim}^A_{\mathcal{L},S}(\lambda) = 1]$$

*is a negligible function. For a static SSE scheme* $\Pi$ *the definition is the same, except $A$ is only allowed search queries in the games.*

Data structures. Our constructions will employ the standard data structures of lists, arrays, and dictionaries. We formalize a dictionary data type in detail because its syntax is relevant to our security analyses. Below, when we say *label,data,* or *data structure*, we mean *bitstring* and will treat them as such in the analysis.

An *dictionary implementation* $\mathsf{Dict}$ consists of four algorithms $\mathsf{Create}, \mathsf{Get}, \mathsf{Insert}, \mathsf{Remove}$. Create takes a list of label-data pairs $(\ell_i, d_i)_{i=1}^m$, where each label is unique, and outputs the data structure $\gamma$. On input $\gamma$ and a label $\ell$, $\mathsf{Get}(\gamma, \ell)$ returns the data item with that label. On input $\gamma$ and $(\ell, d)$, $\mathsf{Insert}(\gamma, (\ell, d))$, outputs an updated data structure, that should contain the new pair. On input $\gamma$ and $\ell$, $\mathsf{Remove}(\gamma, \ell)$ outputs an updated data structure with the pair removed.

We define *correctness* in the obvious way, i.e., the output of $\mathsf{Get}$ is always the data with the (unique) label it is given as input, and that it returns $\perp$ when no data with the label is present.

We say that a dictionary implementation is *history-independent* if for all lists $L$ the distribution of $\mathsf{Create}(L)$ depends only on the members of $L$ and not their order in the list. The $\mathsf{Create}$ algorithm may be randomized or deterministic and satisfy history-independence. A simple way to achieve it is to sort $L$ first, but for large lists there may be more efficient methods.

```
Setup(DB)
1. $K \leftarrow \$ \{0,1\}^\lambda$ allocate list $L$
2. For each $w \in W$:
       $K_1 \| K_2 \leftarrow F(K, w)$
       Initialize counter $c \leftarrow 0$
       For each $\mathsf{id} \in \mathsf{DB}(w)$ :
           $\ell \leftarrow F(K_1, c)$; $d \leftarrow \mathsf{Enc}(K_2, \mathsf{id})$; $c$++
           Add $(\ell, d)$ to the list $L$ (in lex order)
   Set $\mathtt{D} \leftarrow \mathsf{Create}(L)$
3. Output the client key $K$ and $\mathsf{EDB} = \mathtt{D}$.
```

```
Search
Client: On input $(K, w)$,
    $K_1 \leftarrow F(K, 1\|w)$, $K_2 \leftarrow F(K, 2\|w)$
    Send $(K_1, K_2)$ to the server.

Server: On input $\mathsf{EDB} = \mathtt{D}$, message $(K_1, K_2)$,
    For $c = 0$ until Get returns $\perp$,
    $d \leftarrow \mathsf{Get}(\mathtt{D}, F(K_1, c))$; $\mathsf{id} \leftarrow \mathsf{Dec}(K_2, d)$
    Output id
```

Figure 5: Scheme $\Pi_{\mathrm{bas}}$.

We note that we only need the ability to remove data in some limited uses of dictionaries. In all settings were we need a very large dictionary, we can use an add-only version of the data structure.

Extensions and generalization. Two works [4, 14] showed that data structures for single-keyword SSE can be generalized to work for more complex SSE functionalities and models. Specifically, [4] shows how to extend SSE data structures to perform boolean queries on encrypted data (via the OXT protocol), and [14] further extends this functionality to more complex multi-user SSE settings. As a result, all the constructions in this paper can be readily used to support these richer functional settings. All that is needed is to extend the data stored in these data structures from simple document identifiers (in the basic SSE case) to other associated data such as an encrypted key in the case of multi-client SSE (a key used by clients to decrypt documents) or protocol-specific values (such as the '$y$' value in the OXT protocol from [4]). As a consequence, our advancement on the practicality and scale of SSE data structures immediately translates into the ability to support very large and dynamic databases even for functionalities as involved as full boolean SSE search in single- and multi-client SSE settings. We provide concrete evidence of this practical impact in Section 5.5 where we report performance numbers on query execution in these complex settings.

# 3 Static Constructions

Let $\mathsf{Dict} = (\mathsf{Create}, \mathsf{Get}, \mathsf{Insert}, \mathsf{Remove})$ be a dictionary implementation, $F$ be a variable-input-length PRF, and $\Sigma = (\mathsf{Enc}, \mathsf{Dec})$ be a symmetric-key encryption scheme.

Basic Construction. In Figure 5 we give our first and simplest construction, denoted $\Pi_{\mathrm{bas}}$. To build the encrypted database, $\mathsf{Setup}(\mathsf{DB})$ chooses a key $K$ and uses it to derive per-keyword keys for a PRF (to derive pseudorandom labels) and for encryption (to encrypt the identifiers). Then for each keyword $w$, it it iterates over the identifiers in $\mathsf{DB}(w)$. For each identifier, it computes a pseudorandom label by applying the PRF to a counter, encrypts the identifier, and adds the label/ciphertext pair to a list $L$. After all of the results have been processed it builds the dictionary $\mathtt{D}$ from $L$, which becomes the server's index. It is important that $L$ is sorted by the labels before being loaded into the dictionary, or that the dictionary satisfies history independence - Without one of these, the scheme will leak information about the order in which the input was processed.

To search for keyword $w$, the client re-derives the keys for $w$ and sends them to the server, who re-computes the labels and retrieves and decrypts the results.

Leakage function. The leakage function $\mathcal{L}$ for our first construction responds to an initial startup query, and to search queries, where its behavior is defined as follows. We describe the interactive stateful leakage function for the adaptive definitions; The non-adaptive leakage function is the obvious version that iterates over the queries with the adaptive leakage function. On initial input $\mathsf{DB}$, $\mathcal{L}$ outputs $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$,

```
$\underline{\textbf{Init}(\mathsf{DB}, \hat{w}_1, \ldots, \hat{w}_q)}$   // $G_0$, $\boxed{G_1}$
01 $(w_i, (\mathsf{id}_{i,1}, \ldots, \mathsf{id}_{i,S_i}))_{i=1}^m \leftarrow \mathrm{Inv}(\mathsf{DB})$
02 $K \leftarrow_{\$} \{0,1\}^\lambda$ ; $L \leftarrow \varepsilon$
03 for $i \in \{1, \ldots, q\}$ do $\tau_i \leftarrow F(K, \hat{w}_i)$ ; $\boxed{\tau_i \leftarrow T[\hat{w}_i]}$
04 for $j \in \{1, \ldots, m\}$ do
05     $K_1 \| K_2 \leftarrow F(K, w_j)$ ; $\boxed{K_1 \| K_2 \leftarrow T[w_j]}$
06     for $c \in \{1, \ldots S_j\}$ do
07         $\ell \leftarrow F(K_1, c)$ ; $C \leftarrow_{\$} \mathsf{Enc}(K_2, \mathsf{id}_{j,c})$
08         $L \leftarrow L \cup (\ell, C)$
09 $\mathtt{D} \leftarrow \mathsf{Create}(L)$
10 ret $(\mathtt{D}, \tau_1, \ldots, \tau_q)$
```

```
$\underline{\textbf{Init}(\mathsf{DB}, \hat{w}_1, \ldots, \hat{w}_q)}$   // $G_2$, $\boxed{G_3}$
01 $(w_i, (\mathsf{id}_{i,1}, \ldots, \mathsf{id}_{i,S_i}))_{i=1}^m \leftarrow \mathrm{Inv}(\mathsf{DB})$
02 $L \leftarrow \varepsilon$
03 for $i \in \{1, \ldots, q\}$ do $\tau_i \leftarrow T[\hat{w}_i]$
04 for $j \in \{1, \ldots, m\}$ do
05     $K_1 \| K_2 \leftarrow T[w_j]$
06     for $c \in \{1, \ldots S_j\}$ do
07         $\ell \leftarrow F(K_1, c)$ ; $C \leftarrow_{\$} \mathsf{Enc}(K_2, \mathsf{id}_{j,c})$
08         if $w_j \notin \{\hat{w}_1, \ldots, \hat{w}_q\}$ then
09             $\ell \leftarrow U[j, c]$ ; $\boxed{C \leftarrow_{\$} \{0,1\}^{\ell(\lambda)}}$
10         $L \leftarrow L \cup (\ell, C)$
11 $\mathtt{D} \leftarrow \mathsf{Create}(L)$
12 ret $(\mathtt{D}, \tau_1, \ldots, \tau_q)$
```

Figure 6: Games $G_0$–$G_3$ for the proof of Theorem 6. $G_1$ and $G_3$ include the boxed code while $G_0$ and $G_2$ do not. The notation $L \leftarrow L \cup (\ell, C)$ means that $(\ell, C)$ i

saves $\mathsf{DB}$ and an empty list $Q_{\mathrm{srch}}$ as state. Then, for a search input $w$, $\mathcal{L}$ increments $i$, adds $(i, w)$ to $Q_{\mathrm{srch}}$ and outputs $\mathsf{DB}(w)$ and a set $\mathsf{sp}(w, Q_{\mathrm{srch}})$, called the *search pattern for* $w$, defined by

$$\mathsf{sp}(w, Q_{\mathrm{srch}}) = \{j : (j, w) \in Q_{\mathrm{srch}}\}.$$

The search pattern indicates which other queries were also for the keyword $w$, and represents the fact that our scheme will send the same message when a search is repeated.

We deal with non-adaptive $\mathcal{L}$-security first.

**Theorem 6** $\Pi_{\mathsf{bas}}$ *is correct and $\mathcal{L}$-secure against non-adaptive attacks if $F$ is a secure PRF and* $(\mathsf{Enc}, \mathsf{Dec})$ *is RCPA-secure.*

We prove security and correctness separately.

**Proof of security:** We make the simplifying assumption that the adversary never repeats a query because our search protocol is deterministic it is clear that this will not help it. In this case the search pattern is always a singleton, and thus we omit it. The remaining output from the non-adaptive leakage function is $(N, V_1, \ldots, V_q)$, i.e. the size of $\mathsf{DB}$ and the sets of identifiers for each search.

Before giving the simulator we consider games $G_0, \ldots, G_3$ in Figure 6. $G_0$ will compute a distribution identical to $\mathrm{SSENAReal}_\Pi(\lambda)$, $G_3$ will compute a distribution that can be simulated perfectly and the remaining games are hybrids.

The first game $G_0$ responds to a single query to **Init**. It computes $\mathtt{D}$ and the client-message transcripts $\tau_i$ (which each consist of a single message) as specified in the non-adaptive game. It selects a key $K$ and then for each query $\hat{w}_i$ it computes $\tau_i$ as $F(K, \hat{w}_i)$, and then it for each keyword in the chosen $\mathsf{DB}$ it computes the label/ciphertext pairs using $K$ and then creates the dictionary. We have

$$\Pr[G_0] = \Pr[\mathrm{SSENAReal}_\Pi^A(\lambda) = 1]. \quad (1)$$

The next game $G_1$ is like $G_0$ but includes the boxed code. Now every evaluation of $F(K, \cdot)$ is immediately overwritten using the entry from a table $T$. Recall that our convention specifies that when an entry is being accessed for the first time, it is chosen at random and then used thereafter. This means that all of the $\tau_i$ are uniform and independent strings, and similarly the $K_1 \| K_2$ on line 05 are uniform and independent.

We claim there is an efficient adversary $B$ such that

$$\mathbf{Adv}_{F,B_1}^{\mathrm{prf}}(\lambda) = \Pr[G_0] - \Pr[G_1]. \quad (2)$$

This adversary has access to an oracle $\mathbf{Fn}(\cdot, \cdot)$. It runs $A$ to get its **Init** query $(\mathsf{DB}, \hat{w}_1, \ldots, \hat{w}_q)$, and then computes

```
01 $L \leftarrow \varepsilon$
02 $(w_i, (\mathsf{id}_{i,1}, \dots, \mathsf{id}_{i,S_i}))_{i=1}^{m} \leftarrow \mathrm{Inv}(\mathsf{DB})$
03 for $i \in \{1, \dots, q\}$ do $\tau_i \leftarrow \mathbf{Fn}(0, \hat{w}_i)$
04 for $j \in \{1, \dots, m\}$ do
05    $K_1 \| K_2 \leftarrow \mathbf{Fn}(0, w_j)$
06    for $c \in \{1, \dots S_j\}$ do $\ell \leftarrow F(K_1, c)$ ; $C \leftarrow\!\$\, \mathsf{Enc}(K_2, \mathsf{id}_{j,c})$ ; $L \leftarrow L \cup (\ell, C)$
07 $\mathtt{D} \leftarrow \mathsf{Create}(L)$
08 ret $(\mathtt{D}, \tau_1, \dots, \tau_q)$,
```

and it runs $A$ until it halts with some output that $B_1$ uses as its own. It is straightforward to check that $B_1$ is efficient and that

$$\Pr[\mathrm{PRFReal}_F^{B_1}(\lambda) = 1] = \Pr[G_0] \quad \text{and} \quad \Pr[\mathrm{PRFRand}_F^{B_1}(\lambda) = 1] = \Pr[G_1],$$

which gives (2).

Next we consider $G_2$. This game is like $G_1$ except that it deletes some irrelevant code (i.e., the selection of values that are immediately overwritten) and handles unqueried keywords differently, namely in the new code in lines 08 and 09. Those lines implement a change where if $w_j$ was not amongst the queried keywords, then the label $\ell$ is selected at random, using our convention for reading from the possibly-uninitialized table $U$.

We show there exists an efficient adversary $B_2$ such that

$$\mathbf{Adv}_{F,B_2}^{\mathrm{prf}}(\lambda) = \Pr[G_1] - \Pr[G_2]. \tag{3}$$

$B_2$ has access to an $\mathbf{Fn}(\cdot,\cdot)$ oracle, and starts by running $A$ to get its **Init** query $(\mathsf{DB}, \hat{w}_1, \dots, \hat{w}_q)$, and then computes

```
01 $L \leftarrow \varepsilon$
02 $(w_i, (\mathsf{id}_{i,1}, \dots, \mathsf{id}_{i,S_i}))_{i=1}^{m} \leftarrow \mathrm{Inv}(\mathsf{DB})$
03 for $i \in \{1, \dots, q\}$ do $\tau_i \leftarrow T[\hat{w}_i]$
04 for $j \in \{1, \dots, m\}$ do
05    $K_1 \| K_2 \leftarrow T[w_j]$
06    for $c \in \{1, \dots S_j\}$ do
07       $\ell \leftarrow F(K_1, c)$ ; $C \leftarrow\!\$\, \mathsf{Enc}(K_2, \mathsf{id}_{j,c})$
08       if $w_j \notin \{\hat{w}_1, \dots, \hat{w}_q\}$ then $\ell \leftarrow \mathbf{Fn}[j, c]$
09    $L \leftarrow L \cup (\ell, C)$
10 $\mathtt{D} \leftarrow \mathsf{Create}(L)$
11 ret $(\mathtt{D}, \tau_1, \dots, \tau_q)$,
```

finally outputting whatever $A$ outputs. Note that $B_2$ is somewhat unnatural in the way it computes $\ell$ on line 08: If the **Fn** oracle is "real" then it is computing $\ell$ using $F$ with a uniformly random key, and if it is "random" then it is uniformly random string. But in either case, the key $K_1$ is *not* used on line 08 – $\ell$ is always overwritten. We establish (3) by observing that

$$\Pr[\mathrm{PRFReal}_F^{B_2}(\lambda) = 1] = \Pr[G_1] \quad \text{and} \quad \Pr[\mathrm{PRFRand}_F^{B_2}(\lambda) = 1] = \Pr[G_2].$$

This is again straightforward to verify using the observation noted in the previous paragraph. The second case is true as the code syntactically identical, while the first case true even though a fresh random key is used instead of $K_1$ in oracle call on line 08. But since $K_1$ is used nowhere else (and in particular not in line 03), this is equivalent.

Game $G_3$ is exactly like $G_2$ except that it includes the boxed code on line 09 that replaces $C$ with a fresh random string when the keyword $w_j$ is not one of the queried keywords. We show there exists an efficient adversary $B_3$ such that

$$\mathbf{Adv}_{\Pi,B_3}^{\mathrm{ind-rcpa}}(\lambda) = \Pr[G_2] - \Pr[G_3].$$

$B_3$ has access to an oracle $\mathbf{E}$. It runs $A$, and responds to its **Init** query with input $(\mathsf{DB}, \hat{w}_1, \dots, \hat{w}_q)$ by computing

```
01 $L \leftarrow \varepsilon$
02 $(w_i, (\mathsf{id}_{i,1}, \ldots, \mathsf{id}_{i,S_i}))_{i=1}^m \leftarrow \mathrm{Inv}(\mathsf{DB})$
03 for $i \in \{1, \ldots, q\}$ do $\tau_i \leftarrow T[\hat{w}_i]$
04 for $j \in \{1, \ldots, m\}$ do
05    $K_1 \| K_2 \leftarrow T[w_j]$
06    for $c \in \{1, \ldots S_j\}$ do
07       $\ell \leftarrow F(K_1, c)$ ; $C \leftarrow\!\$\, \mathsf{Enc}(K_2, \mathsf{id}_{j,c})$
08       if $w_j \notin \{\hat{w}_1, \ldots, \hat{w}_q\}$ then $\ell \leftarrow U[j, c]$ ; $C \leftarrow \mathbf{E}(j, \mathsf{id}_{j,c})$
09    $L \leftarrow L \cup (\ell, C)$
10 $\mathtt{D} \leftarrow \mathsf{Create}(L)$
11 ret $(\mathtt{D}, \tau_1, \ldots, \tau_q)$.
```

It runs $A$ until it halts and then uses the output of $A$ as its own. We claim that

$$\Pr[\mathrm{RCPAReal}_{\Pi}^{B_3}(\lambda) = 1] = \Pr[G_2] \quad \text{and} \quad \Pr[\mathrm{RCPARand}_{\Pi}^{B_3}(\lambda) = 1] = \Pr[G_3].$$

As with our other adversaries this is only a matter of verifying that the same computation is performed in each case. A subtlety similar to the above case occurs with the usage of $K_2$ in the first equality: When the **if** statement is executed on line 08, $K_2$ is not used as the $\mathbf{E}$ oracle will have an internal key (independent of $K_2$). But since $K_2$ is used no where else, this is equivalent.

In the final step of the proof we give an efficient simulator $S$ such that

$$\Pr[G_3] = \Pr[\mathrm{SSENASim}_{\mathcal{L},S}^{A}(\lambda) = 1]. \tag{4}$$

$S$ takes as input the simplified leakage output $(N, V_1, \ldots, V_q)$, where $N$ is a positive integer and $V_i \subseteq \{0,1\}^\lambda$. It responds to the query by computing

```
$L \leftarrow \varepsilon$
for $i \in \{1, \ldots, q\}$ do
   $\tau_i \leftarrow\!\$\, \{0,1\}^{2\lambda}$ ; $K_1 \| K_2 \leftarrow \tau_i$
   for $c \in \{1, \ldots |V_i|\}$ do
      $\ell \leftarrow F(K_1, c)$ ; $C \leftarrow\!\$\, \mathsf{Enc}(K_2, \mathsf{id}_{j,c})$ ; $L \leftarrow L \cup (\ell, C)$
$N' \leftarrow N - \sum_{i=1}^{Q} |V_i|$
for $j \in \{1, \ldots, N'\}$ do
   $\ell \leftarrow\!\$\, \{0,1\}^\lambda$ ; $C \leftarrow\!\$\, \{0,1\}^{\ell(\lambda)}$ ; $L \leftarrow L \cup (\ell, C)$
$\mathtt{D} \leftarrow \mathsf{Create}(L)$
ret $(\mathtt{D}, \tau_1, \ldots, \tau_q)$.
```

In words, $S$ is first picking each $\tau_i$ to be a random independent string, and then for each $V_i$ it encrypts and labels each identifier using $\tau_i$ (parsed as $K_1 || K_2$) and adds them to $L$. Finally it adds random label/ciphertext pairs to $L$ until there are a total of $N$ pairs and then generates $\mathtt{D}$.

We claim that the distribution of the output of $S$ is identical to that of **Init** in $G_3$, establishing (4). First observe that in both cases the $\tau_i$ are uniform and independent, so we only need to verify that $\mathtt{D}$ has the same distribution conditioned on particular values of $\tau_i$. But then $S$ is computing label/ciphertext values from the same distribution, just in a different order when adding them to $L$ (while $G_3$ adds random label/ciphertexts pairs when looping over each unqueried keyword, $S$ adds them all at once after processing the queried keywords). Since these values are maintained in lexicographic order $\mathtt{D}$ has the same distribution in $G_3$ and when sampled by $S$.

Collecting (1-4), we have

$$\begin{aligned}
\mathbf{Adv}_{\Pi_{\mathrm{bas}},A,S}^{\mathrm{sse-adap}}(\lambda) &= \Pr[\mathrm{SSENAReal}_{\Pi}^{A}(\lambda) = 1] - \Pr[\mathrm{SSENASim}_{\mathcal{L},S}^{A}(\lambda) = 1] \\
&= \Pr[G_0] - \Pr[G_3] \\
&= (\Pr[G_0] - \Pr[G_1]) + (\Pr[G_1] - \Pr[G_2]) + (\Pr[G_2] - \Pr[G_3]) \\
&= \mathbf{Adv}_{F,B_1}^{\mathrm{prf}}(\lambda) + \mathbf{Adv}_{F,B_2}^{\mathrm{prf}}(\lambda) + \mathbf{Adv}_{\Pi,B_3}^{\mathrm{ind-rcpa}}(\lambda).
\end{aligned}$$

By the assumption that $F$ is a secure PRF and $\Pi$ is RCPA secure, this function is negligible. □

**Proof of correctness:** We use the games in Figure 7. The first game $G_0$ implements the game $\mathrm{SSECor}_{\Pi}^{A}(\lambda)$ with the change that, when a label repeats during **Init**, a $\mathsf{bad}$ variable is set and the offending label is replaced with a fresh one. It is straightforward to check that first $G_0$ will only output 1 if $\mathsf{bad}$ is set (i.e., repeated labels are the only source of incorrectness and $G_0$ fixes these), and second that $G_0$ produces an identical distribution to real game when $\mathsf{bad}$ is not set. This gives

$$\mathbf{Adv}_{\Pi,A}^{\mathrm{sse-cor}}(\lambda) = \Pr[\mathrm{SSECor}_{\Pi}^{A}(\lambda) = 1] \leq \Pr[G_0 \text{ sets } \mathsf{bad}]. \tag{5}$$

The next game $G_1$ overwrites evaluations of $F$ with random values stored in $T$ (this happens in both **Init** and $\mathsf{Search}$). There exists an efficient adversary $B_1$ such that

$$\mathbf{Adv}_{F,B_1}^{\mathrm{prf}}(\lambda) = \Pr[G_0 \text{ sets } \mathsf{bad}] - \Pr[G_1 \text{ sets } \mathsf{bad}]. \tag{6}$$

This adversary works very similarly to $B_1$ from the previous proof. We omit the details.

The next game $G_2$ is exactly like $G_1$ except that the labels are chosen at random. We claim there exists an efficient adversary $B_2$ such that

$$\mathbf{Adv}_{F,B_2}^{\mathrm{prf}}(\lambda) = \Pr[G_1 \text{ sets } \mathsf{bad}] - \Pr[G_2 \text{ sets } \mathsf{bad}]. \tag{7}$$

This adversary simulates the games using its oracle in the obvious way, except that it stops after **Init** and declares its output based on if $\mathsf{bad}$ was set. We omit the details, which are straightforward.

Next we claim that

$$\Pr[G_2 \text{ sets } \mathsf{bad}] = \mathrm{negl}(\lambda). \tag{8}$$

This follows from the observation that $\mathsf{Lbls}$ always as polynomial in $\lambda$ number of elements, so there is a negligible probability that a random $\ell$ will be in $\mathsf{Lbls}$. The claim follows from a union bound over the polynomial number of labels $\ell$.

We complete the correctness proof by collecting (5-8):

$$\begin{aligned}\mathbf{Adv}_{\Pi_{\mathrm{bas}},A}^{\mathrm{sse-cor}}(\lambda) &= \Pr[\mathrm{SSECor}_{\Pi}^{A}(\lambda) = 1] = \Pr[G_0 \text{ sets } \mathsf{bad}] \\ &= (\Pr[G_0 \text{ sets } \mathsf{bad}] - \Pr[G_1 \text{ sets } \mathsf{bad}]) + (\Pr[G_1 \text{ sets } \mathsf{bad}] - \Pr[G_2 \text{ sets } \mathsf{bad}]) + \Pr[G_2 \text{ sets } \mathsf{bad}] \\ &= \mathbf{Adv}_{F,B_1}^{\mathrm{prf}}(\lambda) + \mathbf{Adv}_{F,B_2}^{\mathrm{prf}}(\lambda) + \mathrm{negl}(\lambda)\end{aligned}$$

Since $F$ is a PRF, this function is negligible and $\Pi_{\mathrm{bas}}$ satifies the correctness requirement.

Adaptive security in the random oracle model. In the random oracle model we can achieve adaptive security for the same $\mathcal{L}$ if $F$ is replaced with the random oracle $H$ so $F(K,x) := H(K\|x)$, and the encryption algorithm $\mathsf{Enc}$, on inputs $K, m \in \{0,1\}^{\lambda}$, chooses a random $r \in \{0,1\}^{\lambda}$ and outputs $(r, H(K\|r) \oplus m)$. We denote this variant $\Pi_{\mathrm{bas}}^{\mathrm{ro}}$.

**Theorem 7** $\Pi_{\mathrm{bas}}^{\mathrm{ro}}$ *is $\mathcal{L}$-secure against adaptive attacks in the random oracle model.*

**Proof sketch:** This theorem is proved in a similar way to the previous one, except that the simulator programs the random oracle in response to adaptive queries to open the labeled ciphertexts to match the query results as they are revealed. For our version of the PRF and encryption scheme above, the simulator can arrange for the random oracle responses to point at random labels, and for the ciphertexts to decrypt to the revealed results. The only defects in the simulation occur when an adversary manages to query the random oracle with a key before it is revealed, which can be shown to happen with negligible in $\lambda$ probability. □

Alternative approach to adaptive security. We sketch how to modify our protocol to achieve adaptive security without a random oracle at the cost of extra communication. We choose the encryption scheme for the scheme to be of the one-time pad form e.g. CTR mode with a random IV. Now instead of

```
$\underline{\textbf{Init}(\mathsf{DB})}$   // $G_0$, $\boxed{G_1}$, $\boxed{\boxed{G_2}}$
$(w_i, (\mathsf{id}_{i,1}, \ldots, \mathsf{id}_{i,S_i}))_{i=1}^{m} \leftarrow \mathrm{Inv}(\mathsf{DB})$
$K \leftarrow\!\$ \{0,1\}^{\lambda}$ ; $L \leftarrow \varepsilon$ ; $\mathsf{Lbls} \leftarrow \emptyset$
for $j \in \{1, \ldots, m\}$ do
    $K_1 \| K_2 \leftarrow F(K, w_j)$ ; $\boxed{K_1 \| K_2 \leftarrow T[w_j]}$
    for $c \in \{1, \ldots S_j\}$ do
        $\ell \leftarrow F(K_1, c)$ ; $\boxed{\boxed{\ell \leftarrow\!\$ \{0,1\}^{\lambda}}}$ ; $C \leftarrow\!\$ \mathsf{Enc}(K_2, \mathsf{id}_{j,c})$
        if $\ell \in \mathsf{Lbls}$ then
            $\mathsf{bad} \leftarrow \mathsf{true}$ ; $\ell \leftarrow\!\$ \{0,1\}^{\lambda} \setminus \mathsf{Lbls}$
        $\mathsf{Lbls} \leftarrow \mathsf{Lbls} \cup \{\ell\}$
        $L \leftarrow L \cup (\ell, C)$
$\mathtt{D} \leftarrow \mathsf{Create}(L)$
ret$\mathtt{D}$
```

```
$\underline{\mathsf{Search}(\hat{w})}$   // $G_0$, $\boxed{G_1, G_2}$
$K_1 \| K_2 \leftarrow F(K, \hat{w})$ ; $\boxed{K_1 \| K_2 \leftarrow T[w_j]}$
$c \leftarrow 0$ ; $V \leftarrow \emptyset$
while $C \leftarrow \mathsf{Get}(\mathtt{D}, F(K_1, c))$ do
    $\mathsf{id} \leftarrow \mathsf{Dec}(K_2, C)$ ; $V \leftarrow V \cup \{\mathsf{id}\}$
if $V \neq \mathsf{DB}(\hat{w})$ then $\mathsf{win} \leftarrow \mathsf{true}$
ret $K_1 \| K_2$

$\underline{\textbf{Final}()}$   // $G_0, G_1, G_2$
Return $\mathsf{win}$
```

Figure 7: Games $G_0, G_1, G_2$ for the proof of Theorem 6. The notation $L \leftarrow L \cup (\ell, C)$ means that $(\ell, C)$ is added to $L$, in order by $\ell$.

sending the keys $K_1$ and $K_2$, the client computes the labels and encryption pads herself and sends them to the server, who can retrieve the labels and perform the decryption. In general the client will not know when to stop, but we can either have the client retrieve a server-stored encrypted counter first, or have the server send a "stop" message when all of the results have been found. Note that the required additional communication is proportional to the size of the result-set and can overlap the disk access as well as the return of results. Hence, the resulting scheme should perform in practice as good as the prior schemes.

Asymptotic efficiency. The EDB consists of a dictionary holding $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$ identifier/ciphertexts pairs. Searching is fully parallelizeable if the dictionary allows parallel access, as each processor can independently compute $F(K_1, c)$ and retrieve/decrypt the corresponding ciphertext.

Relation to [6] A prior SSE scheme by Chase and Kamara used a dictionary, but in a crucially different way. There, a single label was associated with the entire set $\mathsf{DB}(w)$, and thus security requires padding all of the result sets to the maximum size. We instead associate one label with each result for a keyword (so if there are $T$ documents with a keyword, then there are $T$ independent labels in our scheme but only 1 label in the Chase-Kamara scheme). This allows us to avoid padding and enable parallel searching, resulting in significant storage savings and performance gains on large datasets.

## 3.1 Efficient extensions

We give a sequence of three schemes (denoted $\Pi_{\text{pack}}, \Pi_{\text{ptr}}, \Pi_{\text{2lev}}$, with names explained below) that exhibit the techniques in our most practical $\Pi_{\text{2lev}}$ construction.

Reducing dictionary retrievals: $\Pi_{\text{pack}}$. During a search for $w$, our basic construction performs $|\mathsf{DB}(w)|$ retrievals from the dictionary, each with an independent and random-looking tag. Even an external-memory efficient dictionary will perform relatively poorly when the dictionary is stored on disk.

Most prior schemes suffer from this drawback. To improve locality we modify the basic construction to encrypt several identifiers in each ciphertext. Specifically, we fix a block size $B$, and when building the results list, we process $B$ identifiers at a time and pack them into one ciphertext $d$, with the same tag. We pad the last block of identifiers up to the same length. Searching proceeds exactly as before, except the server decrypts and parses the results in blocks instead of individually. We denote this variant $\Pi_{\text{pack}}$.

This reduces the number of disk accesses from $|\mathsf{DB}(w)|$ dictionary retrievals to $\lceil |\mathsf{DB}(w)|/B \rceil$. We can

```
Setup(DB)
1. $K \leftarrow_{\$} \{0,1\}^\lambda$; allocate array $\mathtt{A}$, list $\mathtt{L}$
2. For each $w \in W$:
      $K_1 \| K_2 \leftarrow F(K,w)$ ; $t \leftarrow \lceil \mathsf{DB}(w)/B \rceil$
      Partition $\mathsf{DB}(w)$ into $B$-blocks $I_1,\ldots,I_t$
      Pad $I_t$ up to $B$ entries if needed
      Choose random empty indices $i_1,\ldots,i_t$ in $\mathtt{A}$
      For $j = 0,\ldots,t$:   //store id blocks in array $\mathtt{A}$
         $d \leftarrow \mathsf{Enc}(K_2, I_j)$;  Store $\mathtt{A}[i_j] \leftarrow d$
      $t' \leftarrow \lceil t/b \rceil$
      Partition $\{i_1,\ldots,i_t\}$ into $b$-blocks $J_1,\ldots,J_{t'}$
      Pad $J_{t'}$ up to $b$ entries if needed
      For $c = 0,\ldots,t'$:   //store ptr blocks in dictionary $\mathtt{D}$
         $\ell \leftarrow F(K_1, c)$ ; $d' \leftarrow \mathsf{Enc}(K_2, J_c)$
         Add $(\ell, d')$ to $\mathtt{L}$
   Set $\mathtt{D} \leftarrow \mathsf{Create}(\mathtt{L})$
3. Output the client key $K$ and $\mathsf{EDB} = (\mathtt{D}, \mathtt{A})$.
```

```
Search
Client: On input $(K, w)$,
      $K_1 \leftarrow F(K, 1\|w)$, $K_2 \leftarrow F(K, 2\|w)$
      Send $(K_1, K_2)$ to the server.

Server: For $c = 0$ until $\mathsf{Get}$ returns $\perp$,
      $d \leftarrow \mathsf{Get}(\mathtt{D}, F(K_1, c))$
      $(i_1,\ldots,i_b) \leftarrow \mathsf{Dec}(K_2, d)$
      For $j = 0,\ldots,b$ (ignore padding indices)
         $m \leftarrow \mathsf{Dec}(K_2, \mathtt{A}[i_j])$
         Parse and output ids in $m$
```

Figure 8: Scheme $\Pi_{\text{ptr}}$.

prove security against non-adaptive or adaptive attacks under the same assumptions, but with the leakage function $\mathcal{L}_B$ that starts by outputting $\sum_{w \in \mathsf{W}} \lceil |\mathsf{DB}(w)|/B \rceil$ instead of $\sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$. We note that this leakage is incomparable to the original leakage (see the discussion at the end of this section).

**Theorem 8** $\Pi_{\text{pack}}$ *is correct and* $\mathcal{L}_B$*-secure against non-adaptive attacks if* $F$ *is a secure PRF and* $(\mathsf{Enc}, \mathsf{Dec})$ *is RCPA-secure.*

This can be proved via a simple extension of the proof for the basic construction. The essential observation is that the server only needs to know how many of the packed blocks to create in the encrypted index. Similar to before, we can achieve adaptive security in the random oracle model or by increasing communication. We omit the details of this analysis due to space and since we will not use $\Pi_{\text{pack}}$ as it is.

FURTHER REDUCTION VIA POINTERS: $\Pi_{\text{ptr}}$. $\Pi_{\text{pack}}$ would be inefficient when returning very large sets $\mathsf{DB}(w)$, as the server still performs $\lceil |\mathsf{DB}(w)|/B \rceil$ dictionary retrievals. Making $B$ large results in too much padding when the dataset contains many keywords only appearing in a few $\ll B$ documents.

We address this tension by modifying the scheme again, calling the new variant $\Pi_{\text{ptr}}$. $\Pi_{\text{ptr}}$ packages the identifiers into encrypted blocks of $B$ as before, but it stores these blocks in random order in external memory and not in the dictionary (technically, we say they are stored in an array). The scheme will now use the dictionary to store encrypted blocks of $b$ pointers to these encrypted blocks. To search, the server will retrieve the encrypted pointers from the dictionary and then follow them to the encrypted identifiers.

$\Pi_{\text{ptr}}$ is described in Figure 8. In this scheme, the $\mathsf{EDB}$ consists of a dictionary $\mathtt{D}$ holding encrypted blocks of $b$ pointers and an array $A$ holding blocks of $B$ encrypted identifiers for a given keyword, where $b$ and $B$ are block size parameters to be chosen. The setup algorithm stores blocks of encrypted results in random locations in $A$, and then stores encrypted pointers to those locations in $\mathtt{D}$, with labels that allow retrieval similar to the prior variants.

We can show that this variant achieves the security for the leakage function $\mathcal{L}_{b,B}$ which initially outputs $\sum_{w \in \mathsf{W}} \lceil |\mathsf{DB}(w)|/B \rceil$ and $\sum_{w \in \mathsf{W}} \lceil |\mathsf{DB}(w)|/(bB) \rceil$, which are the number of blocks in $\mathtt{D}$ and $A$ respectively, and later leakages are just the identifiers as before. We omit this analysis and defer to our main construction.

MOST PRACTICAL VARIANT: $\Pi_{\text{2lev}}$. In real data sets the number of records matched by different keywords will vary by several orders of magnitude. This presents a challenge in optimizing our variants, and we could not find a setting of $B$ and $b$ that gave an acceptable trade-off between index size (due to padding) and search time. Small sets $\mathsf{DB}(w)$ still resulted in a large block of size $B$ in the dictionary and a large block of size $b$ in the array, while huge sets still required many retrievals from the dictionary.

Thus we again modify the scheme to extend the ideas before, calling the new variant $\Pi_{\mathrm{2lev}}$. The crucial difference is that sets $\mathsf{DB}(w)$ can be processed and stored differently based on their sizes, with an extra level of indirection for very large sets that explains the name. Care must be taken to do this with an acceptable form of leakage.

Below we describe the $\Pi_{\mathrm{pack}}$ variant formally. At a high level, it works as follows. It classifies the sets $\mathsf{DB}(w)$ as *small*, *medium*, or *large*. For small sets, it will store the identifiers directly in the dictionary (so no pointers are used, similar to the packed variant $\Pi_{\mathrm{pack}}$). For medium size sets, it will store them as in $\Pi_{\mathrm{ptr}}$, with a block of pointers in the dictionary and then blocks of identifiers in the array. Finally large sets are stored differently, with two levels of indirection: The dictionary is now used to hold pointers that point to blocks of pointers in the array, which point to the identifier blocks.

In $\Pi_{\mathrm{2lev}}$ we again fix parameters $b$ and $B$ to be sizes of blocks in an dictionary $\mathtt{D}$ and array $A$ respectively. The scheme classifies each of the result sets $\mathsf{DB}(w)$ with $|\mathsf{DB}(w)| \leq b$ as *small*, sets of size $b < |\mathsf{DB}(w)| \leq Bb$ as *medium*, and finally sets of size $Bb \leq |\mathsf{DB}(w)| < B^2 b$ as *large*. We will always set $b, B$ so that no set is larger than $B^2 b$.

Small sets fit completely in a block of the top-level dictionary $\mathtt{D}$, and are stored there. Medium sets will be stored as in the previous variant but with a single block of at most $b$ pointers in $\mathtt{D}$ and the corresponding blocks of identifiers in $A$. These sets consist of between $b + 1$ and $bB$ identifiers.

Finally, for large sets we store a block of at most $b$ pointers in $\mathtt{D}$. In each of the $b$ positions pointed to in $A$, we store another block of at most $B$ pointers to other positions in $A$. Finally, these pointers point to blocks of encrypted identifiers. Figure 9 describes the $\mathsf{Setup}(\mathsf{DB})$ function of $\Pi_{\mathrm{2lev}}$ in more detail.

To search, the client works as with the other variants by sending $K_1, K_2$. The server computes the label $\ell \leftarrow F(K_1, 0)$, and retrieves $d \leftarrow \mathsf{Get}(\mathtt{D}, \ell)$, and decrypts $d$ using $K_2$. If it finds identifiers here, then it outputs them and stops. Otherwise, it uses the pointers to retrieve blocks from $A$. If those blocks contain identifiers then it outputs them. Otherwise it follows the next level of pointers to finally find the identifiers, which it decrypts and outputs.

Security. We prove security for the leakage function $\mathcal{L}_{m,b,B}$ that initially outputs $m = |\mathsf{W}|$ and the value

$$S = \sum_{w:|\mathsf{DB}(w)|>b} \lceil |\mathsf{DB}(w)|/B \rceil + \sum_{w:|\mathsf{DB}(w)|>bB} \lceil |\mathsf{DB}(w)|/B^2 \rceil .$$

The value $m$ is the number of data items in $\mathtt{D}$, and the value $S$ is the number of blocks in $A$. This is leaking $S$ itself, which is defined by the above sum, and not the individual summands, resulting leakage that is incomparable to our other variants and to prior schemes. On search queries $\mathcal{L}$ has the same leakage as before.

**Theorem 9** $\Pi_{\mathrm{2lev}}$ *is correct and* $\mathcal{L}_{m,b,B}$*-secure against non-adaptive attacks if* $F$ *is a secure PRF and* $(\mathsf{Enc}, \mathsf{Dec})$ *is RCPA-secure.*

**Proof:** We need to describe a simulator that, given the output of $\mathcal{L}_{m,b,B}$, satifies Definition 5. This means an efficient adversary generates $\mathsf{DB}$ and a list of $q$ queries $\hat{w}_1, \ldots, \hat{w}_q$. Our simulator must generate $\mathsf{EDB}$ and client messages $\tau_1, \ldots, \tau_q$ to simulate search queries.

The high-level approach for the simulator is the same as in the proof of Theorem 6. For each of the queried keywords it will choose random keys $(K_1, K_2)$ to use as the client transcript and then arrange for the correct identifiers for each query to be found in $\mathsf{EDB}$ using those keys. Then it will use the rest of its leakage to pad $\mathsf{EDB}$ with random labels and ciphertexts.

We now describe the simulator. It takes as input $m = |\mathsf{W}|$ and $S$ defined above, and then the identifier sets $V_1, \ldots, V_q$. It initializes a list $\mathtt{L}$ and array $\mathtt{A}$. Then for each of the identifier sets, it chooses $\tau_i$ as two random keys $(K_1, K_2)$ and then mimics the computation for either the small, medium, or large cases in $\Pi_{\mathrm{2lev}}$

```
Setup(DB)
1. $K \leftarrow_\$ \{0,1\}^\lambda$ allocate list $\mathtt{L}$, array $\mathtt{A}$
2. For each $w \in \mathsf{W}$ such that $\mathsf{DB}(w)$ is small ($|\mathsf{DB}(w)| \leq b$):
     $K_1 \| K_2 \leftarrow F(K, w)$ ; $t \leftarrow \lceil |\mathsf{DB}(w)|/B \rceil$
     Encrypt and store a single block of pointers in dictionary $\mathtt{D}$:
        Pad $\mathsf{DB}(w)$ to $b$ elements
        $\ell \leftarrow F(K_1, 0)$ ; $d \leftarrow \mathsf{Enc}(K_2, \mathsf{DB}(w))$
        Add $(\ell, d)$ to $\mathtt{L}$
3. For each $w \in \mathsf{W}$ such that $\mathsf{DB}(w)$ is medium ($b < |\mathsf{DB}(w)| \leq Bb$):
     $K_1 \| K_2 \leftarrow F(K, w)$ ; $t \leftarrow \lceil |\mathsf{DB}(w)|/B \rceil$
     Divide identifiers into blocks, encrypt, and store them array $\mathtt{A}$:
        Partition $\mathsf{DB}(w)$ into $B$-blocks $I_1, \ldots, I_t$ and pad $I_t$ up to $B$ elements if necessary
        Choose random empty indices $i_1, \ldots, i_t$ in $\mathtt{A}$
        For $j = 1, \ldots, t$ do: $d \leftarrow \mathsf{Enc}(K_2, I_j)$ ; $\mathtt{A}[i_j] \leftarrow d$
     Encrypt and store a single block of pointers in dictionary $\mathtt{D}$:
        Pad $\{i_1, \ldots, i_t\}$ to $b$ elements if necessary
        $\ell \leftarrow F(K_1, 0)$ ;   $d' \leftarrow \mathsf{Enc}(K_2, i_1 \| \cdots \| i_b)$
        Add $(\ell, d)$ to $\mathtt{L}$
4. For each $w \in \mathsf{W}$ such that $\mathsf{DB}(w)$ is large ($Bb < |\mathsf{DB}(w)| \leq B^2 b$):
     $K_1 \| K_2 \leftarrow F(K, w)$ ; $t \leftarrow \lceil |\mathsf{DB}(w)|/B \rceil$ ; $t' \leftarrow \lceil t/B \rceil$
     Divide identifiers into blocks, encrypt, and store them array $\mathtt{A}$:
        Partition $\mathsf{DB}(w)$ into $B$-blocks $I_1, \ldots, I_t$ and pad $I_t$ up to $B$ elements if necessary
        Choose random empty indices $i_1, \ldots, i_t$ in $\mathtt{A}$
        For $j = 1, \ldots, t$ do: $d \leftarrow \mathsf{Enc}(K_2, I_j)$ ; $\mathtt{A}[i_j] \leftarrow d$
     Divide pointers into blocks, encrypt, and store them array $\mathtt{A}$:
        Partition $\{i_1, \ldots, i_t\}$ into $b$-blocks $J_1, \ldots, J_{t'}$ and pad $J_{t'}$ to $B$ elements if necessary
        Choose random empty indices $i'_1, \ldots, i'_{t'}$ in $\mathtt{A}$
        For $j = 1, \ldots, t'$ do: $d \leftarrow \mathsf{Enc}(K_2, J_j)$ ; $\mathtt{A}[i'_j] \leftarrow d$
     Encrypt and store a single block of second-level pointers in dictionary $\mathtt{D}$:
        Pad $\{i'_1, \ldots, i'_{t'}\}$ to $b$ elements if necessary
        $\ell \leftarrow F(K_1, 0)$ ;   $d'' \leftarrow \mathsf{Enc}(K_2, i'_1 \| \ldots \| i'_b)$
        Add $(\ell, d'')$ to $\mathtt{L}$
5. $\mathtt{D} \leftarrow \mathsf{Create}(\mathtt{L})$
   Output the client key $K$ and $\mathsf{EDB} = (\mathtt{D}, \mathtt{A})$.
```

Figure 9: Setup for SSE Scheme $\Pi_{\rm 2lev}$.

using $K_1$ and $K_2$. This partially populates $\mathtt{L}$ and $\mathtt{A}$. Finally, the simulator pads $\mathtt{L}$ to contain $m$ entries by adding random label/ciphertext pairs, and it pads $\mathtt{A}$ to $S$ entries by adding random ciphertexts.

Proving this simulator correct proceeds exactly as in the proof of Theorem 6. We omit the numerous details. □

We can prove the following adaptive security theorem either in the random oracle model or by increasing communication. The technique is exactly the same as with $\Pi_{\rm bas}$, but with modifications to program the pointer structure adaptively.

**Theorem 10** $\Pi_{\rm 2lev}$ *is correct and* $\mathcal{L}_{m,b,B}$*-secure against adaptive attacks in the random oracle model.*

Pointers vs. identifiers. Although pointers are smaller than identifiers in our implementations, $\Pi_{\rm 2lev}$ packs the same number of pointers or identifiers together ($b$ in the dictionary, or $B$ in the array) to simplify the presentation. The actual implementation packs more pointers into a block than identifiers. Formally, we introduce parameters $b', B'$, and modify $\Pi_{\rm 2lev}$ as follows.

- When storing identifiers in the dictionary (in the small case), it packs up to $b$ of them together, but when storing pointers there it packs $b'$ in the same amount of space.
- When storing identifiers in the array (in the medium and large cases), it packs up to $B$ of them together, but when storing pointers there it packs $B'$ together in the same amount of space.

This causes an analogous change to the value $S$ leaked, which can be calculated similarly. We omit the formal analysis (which is almost identical to that of $\Pi_{\mathrm{2lev}}$).

Leakage discussion. The leakage functions $\mathcal{L}_B, \mathcal{L}_{b,B}, \mathcal{L}_{m,b,B}$ are non-standard. First consider $\mathcal{L}_B$, and how it compares to $\mathcal{L}$ which outputs $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$. Any input $\mathsf{DB}$ with $m$ unique keywords, each with $|\mathsf{DB}(w)| \leq b$, will be indistinguishable under $\mathcal{L}_B$, but many of them will not be under $\mathcal{L}$. A similar incomparability goes in the other direction. We are not aware of a scenario where this difference is important for reasonably small $B$. The function $\mathcal{L}_{b,B}$ leaks strictly more information than $\mathcal{L}_B$ (actually $\mathcal{L}_b$), but it also does not appear to be harmful. Finally, $\mathcal{L}_{m,b,B}$ leaks this type of size information *and* the number of keywords $m$. The number $m$ seems to be the most useful information for an adversary, but in prior work it has been considered acceptable. It is possible to modify the scheme to avoid leaking exactly $m$, say by storing blocks of a different size in the dictionary.

# 4 Dynamic Constructions

We extend our static SSE constructions to support changes to the database. Our dynamic SSE constructions will consist of a statically encrypted database $\mathsf{EDB}$ using any of the schemes described above, and an auxiliary encrypted database $\mathsf{EDB}^{+}$ which is maintained to be of the form of a basic dictionary-based scheme. The $\mathsf{EDB}^{+}$ is initially empty and changes as updates happen.

Add-only scheme: $\Pi^+_{\mathrm{bas}}$. We start with an extension of $\Pi_{\mathrm{bas}}$, denoted $\Pi^+_{\mathrm{bas}}$ that supports additions only, meaning $\mathsf{add}, \mathsf{edit}^+$ inputs from the client during $\mathsf{Update}$. $\Pi^+_{\mathrm{bas}}$ is simpler and possibly interesting in its own right.

To support additions we use a dictionary $\mathtt{D}^+$ which is initially empty and to which a pair $(\ell, d)$ is added with each keyword addition; here $\ell$ is a label computed from the keyword and a keyword-specific counter, and $d$ is the encryption of the record $\mathsf{id}$ involved in the addition operation. Search for a keyword $w$ is performed by the server by first searching $\mathtt{D}$ as in the static case, then re-computing all labels corresponding to $w$ in $\mathtt{D}^+$. The latter labels are computed using a $w$-specific key provided by the client and a running counter.

Note that addition operations involving keyword $w$ require the client to know the current value of the $w$-specific counter. For this, the scheme maintains a dictionary $\mathtt{D}_{\mathrm{count}}$ associating each keyword that was ever added via $\mathsf{edit}^+$ or $\mathsf{add}$ with its current counter value. $\mathtt{D}_{\mathrm{count}}$ can be stored at the client or stored at the server and retrieved by the client for performing update operations. We formalize a scheme $\Pi^+_{\mathrm{bas}}$ where the client stores locally the dictionary $\mathtt{D}_{\mathrm{count}}$ and discuss below a stateless variant. We assume throughout that the client never tries to add a record/keyword pair that is already present - it is easy, but messy, to extend our scheme and the leakage profile to handle this.

In $\Pi^+_{\mathrm{bas}}$, $\mathsf{Setup}(\mathsf{DB})$ is exactly as in $\Pi_{\mathrm{bas}}$ except that the client also initializes $\mathtt{D}_{\mathrm{count}}$ to be an empty dictionary and keeps it as state, and the server initializes an empty dictionary $\mathtt{D}^+$ that is stored with $\mathsf{EDB}$. We also modify the scheme to save an additional key $K^+$ (which could be derived to save space). We next give the update protocol.

Update: We only specify the protocol with client input $\mathsf{op} \in \{\mathsf{add}, \mathsf{edit}^+\}$. The parties work exactly the same on either type of operation. To update the client has input $\mathsf{id}, \mathsf{W}_{\mathsf{id}}$.and proceeds as follows:

```
For $w \in \mathsf{W}_{\mathsf{id}}$:
    $K_1^+ \| K_2^+ \leftarrow F(K^+, w)$
    $c \leftarrow \mathsf{Get}(\mathtt{D}_{\mathrm{count}}, w)$; If $c = \bot$ then $c \leftarrow 0$
    Set $\ell \leftarrow F(K_1^+, c)$ ; $d \leftarrow \mathsf{Enc}(K_2^+, \mathsf{id})$
    $c{+}{+}$ ; Insert $(w, c)$ into $\mathtt{D}_{\mathrm{count}}$
```

```
        Add $(\ell, d)$ to $L$ in lexicographic order
    Send $L$ to the server.
```

When inserting $(w, c)$ into $\mathtt{D}_{\text{count}}$, we assume that it will overwrite any previous entry $(w, \cdot)$ if it exists.

Finally, the server adds each $(\ell, d) \in L$ to $\mathtt{D}^+$. This completes the update protocol.

To complete $\Pi^+_{\text{bas}}$ we describe the protocol Search.

Search: On input $w$, the client computes $K_1 \| K_2 \leftarrow F(K, w)$; $K_1^+ \| K_2^+ \leftarrow F(K^+, w)$ and send $(K_1, K_2, K_1^+, K_2^+)$ to the server.

Upon receiving the message, the server computes its output as follows:

```
For $c = 0$ until Get returns $\perp$,
    $d \leftarrow \mathsf{Get}(\mathtt{D}, F(K_1, c))$ ; $\mathsf{id} \leftarrow \mathsf{Dec}(K_2, d)$
    Output each id
For $c = 0$ until Get returns $\perp$,
    $d \leftarrow \mathsf{Get}(\mathtt{D}^+, F(K_1^+, c))$ ; $\mathsf{id} \leftarrow \mathsf{Dec}(K_2^+, d)$
    Output each id
```

Intuitively, the server is repeating the search procedure from $\Pi_{\text{bas}}$ twice: Once with $(K_1, K_2)$ and $\mathtt{D}$, and then with $(K_1^+, K_2^+)$ and $\mathtt{D}^+$.

Leakage profile for $\Pi^+_{\text{bas}}$. Let us first give some intuition for the leakage of $\Pi^+_{\text{bas}}$. Initially the leakage is exactly like $\Pi_{\text{bas}}$, where only the size of DB is leaked. Upon an $\mathsf{edit}^+$ or add query, if the keywords being added were not previously searched for, then the server learns nothing other than number of record/keyword pairs added (not even the if the operation was $\mathsf{edit}^+$ vs. add). If, however, one (or more) of the keywords were previously searched for, then the server can reuse its keys from before to detect the presence of these keywords (this type of leakage is inherent when the keys provided to the server for searching are deterministically generated and the same each time). The leakage on a search is similar to before, except now for record/keyword pairs in $\mathtt{D}^+$ the server can recognize when they were added. The order for pairs in $\mathtt{D}$ generated at setup time is still hidden, however.

We proceed with the formal definition of $\mathcal{L}^+$ for adaptive security. Amongst its state, it will keep a list $Q$ describing all queries issued so far, where an entry of $Q$ is of the form $(i, \mathsf{op}, \ldots)$, meaning a counter, the operation type, and then the one or more inputs to the operation.

On initial input DB, $\mathcal{L}^+$ creates a state consisting of a counter $i \leftarrow 0$, an empty list $Q$ and DB, and a set ID initialized to contain all of the identifiers in DB. Let us define the *search pattern* $\mathsf{sp}(w, Q)$ of a keyword with respect to $Q$ to be the indices of queries that searched for the keyword $w$, i.e.

$$\mathsf{sp}(w, Q) = \{j : (j, \mathsf{srch}, w) \in Q\}.$$

For an identifier id and keyword $w$, the *add pattern of* $\mathsf{id}, w$ *with respect to* $Q$ is the indices that added $w$ to the document id, i.e.

$$\mathsf{ap}(\mathsf{id}, w, Q) = \{j : (j, \mathsf{add}, \mathsf{id}, \mathsf{W}_{\mathsf{id}}) \in Q, w \in \mathsf{W}_{\mathsf{id}}\} \cup \ \{j : (j, \mathsf{edit}^+, \mathsf{id}, \mathsf{W}_{\mathsf{id}}) \in Q, w \in \mathsf{W}_{\mathsf{id}}\}.$$

Finally, we let the *add pattern of keyword* $w$ *with respect to* $Q$ *and* ID be the set of all identifiers to which $w$ was ever added (via a add or $\mathsf{edit}^+$ operation) along with the indices showing when they were added. That is,

$$\mathsf{AP}(w, Q, \mathsf{ID}) = \{(\mathsf{id}, \mathsf{ap}(\mathsf{id}, w, Q)) : \mathsf{id} \in \mathsf{ID}, \mathsf{ap}(\mathsf{id}, w, Q) \neq \emptyset\}.$$

$\mathcal{L}^+$ produces outputs for the initial query, $\mathsf{edit}^+$ and add updates, and search queries as follows:

- On initial input DB it saves state as defined above and outputs $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$.
- For a search query $w$, $\mathcal{L}^+$ appends $(i, \mathsf{srch}, w)$ to $Q$ and increments $i$. Then it outputs $\mathsf{sp}(w, Q)$, $\mathsf{DB}(w)$, and $\mathsf{AP}(w, Q, \mathsf{ID})$.
- Update queries for $\mathsf{edit}^+$ and add operations are handled similarly. For a query $(\mathsf{edit}^+/\mathsf{add}, \mathsf{id}, \mathsf{W}_{\mathsf{id}})$, $\mathcal{L}^+$ first appends $(i, \mathsf{edit}^+/\mathsf{add}, \mathsf{id}, \mathsf{W}_{\mathsf{id}})$ to $Q$, adds id to ID, and increments $i$. It outputs $|\mathsf{W}_{\mathsf{id}}|$ and the

(lexicographically ordered) set of search patterns

$$\{\mathsf{sp}(w, Q) : w \in \mathsf{W}_{\mathsf{id}}\}.$$

If any of the search patterns was non-empty, then it also outputs $\mathsf{id}$.

While subtle in its formulation, $\mathcal{L}^{+}$ is essentially the best possible leakage for an SSE scheme that generates the same search keys on repeated searches.

In words, the search query leakage includes $\mathsf{sp}(w, Q)$ and $\mathsf{DB}(w)$ for obvious reasons. The add pattern of $w$, $\mathsf{AP}(w, Q, \mathsf{ID})$, is the set of $\mathsf{id}$ matching $w$ added later along with "history" information $\mathsf{ap}(\mathsf{id}, w, Q)$ indicating when they added. The order information represents that the server can look at $\mathtt{D}^{+}$ and see when each $\mathsf{id}$ was added by rewinding and re-running searches. For updates $\Pi^{+}_{\mathsf{bas}}$ leaks *only the size of the update* if the added keywords have not been searched for. If any of them have been searched for, then the server learns that "a keyword with search pattern $\mathsf{sp}(w, Q)$ was added" via the set of search patterns in the update leakage. Finally it learns the $\mathsf{id}$ being updated because it has the ability to search for any of its keywords. Each of these leakage components is unavoidable for a deterministic SSE scheme, and we regard them as minimal.

**Theorem 11** *$\Pi^{+}_{\mathsf{bas}}$ is correct and $\mathcal{L}^{+}$-secure against non-adaptive attacks if $F$ is a secure PRF and $(\mathsf{Enc}, \mathsf{Dec})$ is RCPA-secure.*

**Proof sketch:** We briefly describe the required simulator; it can be shown correct via the same type of proof used for Theorem 6. The simulator builds the initial $\mathsf{EDB}$ exactly as in that proof, and initially sets $\mathtt{D}^{+}$ to empty. To answer search queries it selects random keys $K_1, K_2, K_1^{+}, K_2^{+}$, with repetitions as described by the search pattern.

Finally we need simulate update queries. More precisely, the simulator needs to simulate the message sent by the client, which consists of several label/data pairs. The simulator must decide for each pair sent if it is supposed to be random (and meaningless) or if the pair should be computed with one of the keys used for a search query transcript. It does this using both the add pattern leakage from the search queries and the leakage from update queries which include the $\mathsf{id}$ to encrypt when the addition includes a keyword that was previously searched.

Checking that the simulator is correct involves the same techniques as the proof of Theorem 6. We omit these many details, which are all small extensions of that proof. □

STATELESS CLIENT VARIANT. Above, the client keeps a dictionary $\mathtt{D}_{\text{count}}$ containing one counter per keyword that is added after initialization. We could modify the scheme so that the client is stateless by storing $\mathtt{D}_{\text{count}}$ in encrypted form at the server and having the client download and re-encrypt *all of* $\mathtt{D}_{\text{count}}$ for each update (note that the size of $\mathtt{D}_{\text{count}}$ is as the number of distinct keywords added via $\mathsf{add}$ and $\mathsf{edit}^{+}$ and *not* the total number of keywords in the set $\mathsf{W}$). In this variant the server will learn how many *new* keywords are added each time by watching if $\mathtt{D}_{\text{count}}$ grows.

DYNAMIC SCHEME $\Pi^{\text{dyn}}_{\text{bas}}$. We now augment the $\Pi_{\text{bas}}$ scheme with $\mathsf{del}$ and $\mathsf{edit}^{-}$ operations to obtain our fully dynamic scheme $\Pi^{\text{dyn}}_{\text{bas}}$. We will implement deletions by maintaining a revocation list and having the server discard results that have been revoked.

To delete a record/keyword pair $(\mathsf{id}, w)$ from the server's storage, the client will generate a pseudorandom *revocation identifier* and send it to the server. During searches, the client will give the server a key that allows it to recompute revocation identifiers, which it will then use to filter out deleted results. This complicates our addition protocols. To add a pair that was previously deleted, the protocol must "unrevoke" that pair by having the server delete its revocation identifier.

We now formally specify $\Pi^{\text{dyn}}_{\text{bas}}$. $\mathsf{Setup}$ is exactly the same as $\Pi^{+}_{\text{bas}}$, except that the server also initializes an empty *set* $S_{\text{rev}}$. As a data structure, $S_{\text{rev}}$ will support additions, deletions, and membership testing. The scheme now keeps an additional key $K^{-}$, which can be derived from a master key to save space.

$\underline{\mathsf{Update}}$: We first describe how to handle client inputs with $\mathsf{op} \in \{\mathsf{del}, \mathsf{edit}^{-}\}$. The client takes as input $(\mathsf{del}/\mathsf{edit}^{-}, \mathsf{id}, \mathsf{W}_{\mathsf{id}})$, and then computes

```
For $w \in \mathsf{W}_{\mathsf{id}}$ do
    $K_1^- \leftarrow F(K^-, w), \mathsf{revid} \leftarrow F(K_1^-, \mathsf{id})$
    Add $\mathsf{revid}$ to $L_{\text{rev}}$ in lexicographic order
Send $L_{\text{rev}}$ to the server.
```

The server receives $L_{\text{rev}}$ and adds each $\mathsf{revid}$ to $S_{\text{rev}}$. This completes $\mathsf{Update}$ for the $\mathsf{del}$ and $\mathsf{edit}^-$ operations.

Next we define $\mathsf{Update}$ for $\mathsf{op} \in \{\mathsf{add}, \mathsf{edit}^+\}$. On input $(\mathsf{add}/\mathsf{edit}^+, \mathsf{id}, \mathsf{W}_{\mathsf{id}})$, the client performs a computation similar to the list $L$ computation in $\Pi_{\text{bas}}^+$, except that it also includes the appropriate $\mathsf{revid}$ values. It then awaits a response from the server specifying which additions resulted in a true addition and which caused an "unrevocation", and uses this information to increment the correct counters. In code, the client does the following:

```
For $w \in \mathsf{W}_{\mathsf{id}}$:
    $K_1^+ \| K_2^+ \leftarrow F(K^+, w)$ ; $K_1^- \leftarrow F(K^-, w)$
    $c \leftarrow \mathsf{Get}(\mathtt{D}_{\text{count}}, w)$; If $c = \perp$ then $c \leftarrow 0$
    $\ell \leftarrow F(K_1^+, c)$ ; $d \leftarrow \mathsf{Enc}(K_2^+, \mathsf{id})$
    $\mathsf{revid} \leftarrow F(K_1^-, \mathsf{id})$
    Add $(\ell, d, \mathsf{revid})$ to $L$ in lexicographic order
Send $L$ to the server.
```

The server generates its response $r \in \{0,1\}^{|L|}$ as follows. For the $i$-th pair $(\ell, d, \mathsf{revid}) \in L$ in order, if $\mathsf{revid} \in S_{\text{rev}}$, it sets the $i$-th bit of $r$ to 1 and deletes $\mathsf{revid}$ from $S_{\text{rev}}$. Else, it clears that bit to 0 and adds $(\ell, d)$ to $\mathtt{D}$. Finally, it sends $r$ to the client.

Now the client increments the counters for keywords corresponding to 0 bits in $r$. It processes the keywords $w \in \mathsf{W}_{\mathsf{id}}$ in order of their labels in $L$. For the $i$-th keyword $w$ in that order, if the $i$-th bit of $r$ is 0 it computes $c \leftarrow \mathsf{Get}(\mathtt{D}_{\text{count}}, w)$, increments $c$, and inserts $(w, c)$ into $\mathtt{D}_{\text{count}}$. This completes the update protocol.

The last component of $\Pi_{\text{bas}}^{\text{dyn}}$ is the search protocol.

Search: On client input $w$, it sets $K_1^- = F(K^-, w)$, and then computes $(K_1, K_2, K_1^+, K_2^+)$ as in $\Pi_{\text{bas}}^+$. It sends $(K_1, K_2, K_1^+, K_2^+, K_1^-)$ to the server. The server computes the result identifiers using the first four keys exactly as in $\Pi_{\text{bas}}^+$, except before outputting each $\mathsf{id}$ it computes $\mathsf{revid} = F(K_1^-, \mathsf{id})$ and tests if $\mathsf{revid} \in S_{\text{rev}}$. If so, it discards $\mathsf{id}$ instead of outputting it.

Leakage function. We now define the leakage profile $\mathcal{L}_{\text{dyn}}$. It will maintain a list of query information $Q$ and set of identifiers $\mathsf{ID}$ like $\mathcal{L}^+$ from above. Below we use the same definitions for $\mathsf{sp}, \mathsf{ap}, \mathsf{AP}$ as in $\mathcal{L}^+$, and define the following analogous patterns $\mathsf{dp}, \mathsf{DP}$ for deletions:

$$\mathsf{dp}(\mathsf{id}, w, Q) = \{j : (j, \mathsf{del}, \mathsf{id}, \mathsf{W}_{\mathsf{id}}) \in Q, w \in \mathsf{W}_{\mathsf{id}}\} \cup \{j : (j, \mathsf{edit}^-, \mathsf{id}, \mathsf{W}_{\mathsf{id}}) \in Q, w \in \mathsf{W}_{\mathsf{id}}\}.$$

and

$$\mathsf{DP}(w, Q, \mathsf{ID}) = \{(\mathsf{id}, \mathsf{dp}(\mathsf{id}, w, Q)) : \mathsf{id} \in \mathsf{ID}, \mathsf{dp}(\mathsf{id}, w, Q) \neq \emptyset\}.$$

Intuitively, $\mathsf{dp}(\mathsf{id}, w, Q)$ is the set of indices of queries that deleted $w$ from $\mathsf{id}$, and $\mathsf{DP}(w, Q, \mathsf{ID})$ is the set of identifiers form which $w$ was deleted, along with the corresponding deletion pattern.

- On first input $\mathsf{DB}$, $\mathcal{L}_{\text{dyn}}$ initializes a counter $i \leftarrow 0$, empty list $Q$, set $\mathsf{ID}$ to be identifiers in $\mathsf{DB}$. It saves $\mathsf{DB}, i, \mathsf{ID}, Q$ as state, and outputs $N = \sum_{w \in \mathsf{W}} |\mathsf{DB}(w)|$.
- On search input $w$, $\mathcal{L}_{\text{dyn}}$ appends $(i, \mathsf{srch}, w)$ to $Q$, increments $i$, and outputs $\mathsf{sp}(w, Q)$, $\mathsf{DB}(w)$, $\mathsf{AP}(w, Q, \mathsf{ID})$, and $\mathsf{DP}(w, Q, \mathsf{ID})$.
- On update input $(\mathsf{add}/\mathsf{edit}^+, \mathsf{id}, \mathsf{W}_{\mathsf{id}})$, it appends $(i, \mathsf{add}/\mathsf{edit}^+, \mathsf{id}, \mathsf{W}_{\mathsf{id}})$ to $Q$, adds $\mathsf{id}$ to $\mathsf{ID}$, and increments $i$. It outputs $\mathsf{add}, |\mathsf{W}_{\mathsf{id}}|$ and the set
$$\{(\mathsf{sp}(w, Q), \mathsf{ap}(\mathsf{id}, w, Q), \mathsf{dp}(\mathsf{id}, w, Q)) : w \in \mathsf{W}_{\mathsf{id}}\}.$$
Finally, if any of the $\mathsf{sp}(w, Q)$ are non-empty, then it also outputs $\mathsf{id}$.

- On update input $(\mathsf{del}/\mathsf{edit}^-, \mathsf{id}, \mathsf{W}_{\mathsf{id}})$, it appends $(i, \mathsf{del}/\mathsf{edit}^-, \mathsf{id}, \mathsf{W}_{\mathsf{id}})$ to $Q$, adds $\mathsf{id}$ to $\mathsf{ID}$, and increments $i$. Then it computes its output exactly as in the $\mathsf{add}/\mathsf{edit}^+$ case above, except that it outputs $\mathsf{del}$ instead of $\mathsf{add}$ as the first component.

The leakage on searches is minimal: It consists of all patterns of searches, deletions, and additions that can be derived once the server has the ability to search for a keyword and rewind the database. For leakage on updates, the server will learn when/if that identifier has had the same keywords added or deleted before, and also when/if the same keywords have been searched for. This comes from observing the $\mathsf{revid}$ values, which will repeat every time the same identifier/keyword pair is added or deleted. Note that, if same keyword is added/deleted from two documents, then this information is not leaked until it is searched for (contrast this with [15] which leaks this information always).

We have the following theorem.

**Theorem 12** $\Pi^{\mathrm{dyn}}_{\mathrm{bas}}$ *is correct and* $\mathcal{L}_{\mathrm{dyn}}$*-secure against non-adaptive attacks if* $F$ *is a secure PRF and* $(\mathsf{Enc}, \mathsf{Dec})$ *is RCPA-secure.*

**Proof sketch:** The simulator is very similar to the one for the add-only version of the scheme. It chooses the keys $K_1, K_2, K_1^+, K_2^+, K_1^-$ for each search at random, which repetitions specified by the search pattern, and it sets up the initial $\mathsf{EDB}$ in the same way. It simulates addition instructions in exactly the same way (note that the server message sent back to the client is deterministically computable from the server state and does not need to be simulated). To simulate deletions, we observe that they are exactly like additions but without additional ciphertexts to associate with labels, so the leakage (which is the same as in additions) is easily enough to give a consistent simulation. □

Asymptotic Analysis. To add a file the client sends one label/ciphertext/$\mathsf{revid}$ per record/keyword pair being changed. For deletions, the $\mathtt{D}_{\mathrm{count}}$ dictionary is not involved. The client just sends one $\mathsf{revid}$ per document/keyword to be deleted. Assuming the dictionaries $\mathtt{D}$, $\mathtt{D}^+$, and the revocation list are fully read-parallel, and the number of deletions is much smaller than the size of the EDB, each search operation continues to have the same order run-time complexity as in the basic static construction of Figure 5.

Discussion and comparison to prior work. Our scheme $\Pi^{\mathrm{dyn}}_{\mathrm{bas}}$ is unsatisfying in some situations as it does not reclaim space after deletions. While this is a drawback, all known dynamic SSE schemes [15,16,22] have severe drawbacks in different dimensions, and no scheme has achieved an ideal combination of leakage, index size, and full functionality like reclaiming space.

The scheme of [22] has no security proof, and the scheme of [15] has a worst-case quadratic size encrypted index. The dynamic scheme in [16] has much more leakage than our scheme, effectively leaking the pattern of all intersections of everything that is added or deleted, whether or not the keywords were searched for. For an example, suppose $\{w_1, w_2, w_3\}$ are added to $\mathsf{id}_1$, $\{w_1, w_2\}$ are added to $\mathsf{id}_2$, and $\{w_1\}$ is added to $\mathsf{id}_3$. Then [16] will leak that exactly one common keyword was added to all three and that exactly two common keywords were added to the first two (but not the third) and so on. This structural "equality pattern" is the sort of leakage that we do not leak.

Not reclaiming space allows our implementations to be much simpler and also gives us the flexibility to apply various efficiency optimizations (as in section 3 A) to the static scheme which seem hard to achieve when in-place updates have to be supported. As our data structures are more compact than prior work, the overall space requirements will be lower anyway for some number of deletes. In particular, as compared to prior work [15] we are not forced to estimate an upper bound (by necessity, conservative) on the maximum database size.

In some settings where SSE is used as a component, the encrypted database is re-encrypted for security reasons [14]. In these settings we can reclaim space and combine the auxiliary data structure with the main static data structure while re-encrypting.

Application to $\Pi_{\mathrm{ptr}}, \Pi_{\mathrm{pack}}, \Pi_{\mathrm{2lev}}$, and adaptive variants. The dynamic extensions to $\Pi_{\mathrm{bas}}$ can be applied as-is to other variants, resulting in almost the same leakage $\mathcal{L}_{\mathrm{dyn}}$. The only difference is the size leakage in the initial input $\mathsf{DB}$, which changes according to the different schemes. In our implementation in the next section we consider these variants. We can also achieve adaptive security in the random oracle model via the same techniques used for the static constructions.

# 5 Implementation

We report on our implementations of $\Pi_{\mathrm{2lev}}$ and $\Pi_{\mathrm{pack}}$ (described in Section 3), with extensions for dynamic data updates (Section 4). The former scheme is the most efficient and scales to the largest datasets; it represents our current prototype. The latter is a simplification of the original OXT implementation which we introduced in [4] and is discussed here to better illustrate the effectiveness of the ideas in $\Pi_{\mathrm{2lev}}$ and the improvement over prior work.

Practical criteria. Before describing our results, we enumerate some of the practical criteria that we optimize for in the $\Pi_{\mathrm{2lev}}$ prototype.

- Parallel EDB access: The server should be able to issue concurrent access requests to EDB when processing a search. Modern disk controllers handle thousands of concurrent requests and optimize disk access patterns, increasing transfer bandwidth by orders of magnitude when compared with sequential access. Requests are served out-of-order but the performance benefits offset the additional implementation complexity.
- EDB goodput: EDB design should maximize I/O goodput, i.e., the ratio of data used by the processing of a query relative to the total amount of data retrieved from external storage. In addition to selecting an appropriate dictionary, we achieve this by setting the parameters $b, b', B, B'$ in $\Pi_{\mathrm{2lev}}$ to take maximum advantage of the block device.
- Small EDB storage: The dictionary used in EDB should minimize storage overhead while satisfying the other constraints.
- Lightweight EDB updates: Update information will be independent from the EDB and implemented in-memory. This is consistent with our envisioned scenarios where updates are either infrequent or periodically folded into the main data structure via re-encryption of the entire database.

Input datasets. Our implementation accepts as input both relational databases and document collections. The latter are mapped to relational database tables with document attributes, such as author name, creation date, etc., stored in atomic columns and with the document content stored in a text column.

We target clear-text datasets (DBs) that consist of several tens of billions of distinct (keyword, id) pairs. The EDBs generated from such datasets take several terabytes of storage and require several times more temp storage for Setup. We aim to process such datasets efficiently (Setup(DB) and Search) on medium size 64-bit x86 platforms (in our configuration, 8 cores, 96GB of RAM, and $\approx$ 100TB RAID volume on external storage box).

The constructions described in this paper and their implementations can be extended to support richer functional settings than simple keyword search, such as SSE in multi-client settings or boolean queries via the OXT protocol [4] (see end of Section 2), by storing in the EDB for each (keyword, id) pair more data than just the encrypted document id. In the following, we use the term *tuple* for the data stored per (keyword, id) pair in any of these functional settings.

Organization. The next two subsections describe our experiences with the $\Pi_{\mathrm{pack}}$ prototype, which is the subset of the OXT implementation [4] relevant to this work, and the design and implementation of our $\Pi_{\mathrm{2lev}}$ (see Figure 9). A particular challenging issue for both prototypes was EDB generation time; the Setup implementation for $\Pi_{\mathrm{2lev}}$ is discussed separately in Section 5.3. Section 5.4 describes how these constructs are used to support richer functional settings, such as OXT. Finally, Section 5.5 describes several representative experiments.

### 5.1 $\Pi_{\text{pack}}$ Implementation

The discussion of the $\Pi_{\text{pack}}$ implementation here is intended as a preamble to our presentation of $\Pi_{\text{2lev}}$ in the next subsection as it serves to motivate the optimizations applied to the latter construction. Our implementation of $\Pi_{\text{pack}}$ instantiates the EDB dictionary using a bucket hash table. Buckets are split in equal-size *locations*, which are used to store equal-size groups of tuples created by partitioning the $\mathsf{DB}(w)$ sets. The location size is equal to the group size plus the size of the attached label. Each group can be stored in any of the free locations in the bucket determined by hashing its label. As usual, the hash map is over-sized to allow for all groups to be placed successfully; empty locations are filled with random bits to mask the total number of groups in the EDB.

Using a bucket hash for the dictionary allowed us to avoid sorting the tuples by label (as required for security) before creating the dictionary. This worked by ensuring the dictionary is *history independent*, meaning the output of $\mathsf{Create}(L)$ depends only on the members of $L$ and not on the order they were added to $L$.

The bucket hash table is stored in one large file on an ext4 RAID partition of attached storage. The bucket size is set to a multiple of the RAID stripe size[1], and buckets are aligned with the pages of the underlying file system.

The two most significant drawbacks with the above construction are the need for oversizing the hash table, which translates into a much larger EDB than needed, and the poor goodput, as one have to retrieve an entire bucket to access a group of tuples. In experiments with configurations and data sets similar to those described in [4], the hash table has a load factor of $\approx 60\%$ (i.e., over-sized by a factor of $\approx 1.6$) for the placement to be successful, and goodput is $\approx 1\%$, as there are 96 locations per bucket.

To achieve a higher load factor (smaller EDB), we built another $\Pi_{\text{pack}}$ prototype which uses a Cuckoo Hash (CH) table modeled after [9] for the dictionary; page size and alignment are the same as for the bucket hash dictionary in the previous construction. Although we achieve load factors a little over 90%, the cost of handling collisions during EDB generation is very high. Moreover, making the dictionary *history independent* is much more difficult when using a CH table and likely inefficient in our setting.

We designed a more efficient algorithm to handle collisions during EDB generation, which leverages the server memory, but we found its performance to be limited by its database access patterns (see Section 5.5). Finally, the need to improve the goodput motivated the design of $\Pi_{\text{ptr}}$ and $\Pi_{\text{2lev}}$.

### 5.2 $\Pi_{\text{2lev}}$ Implementation

In order to meet the criteria stated at the beginning of this section and avoid the drawbacks of $\Pi_{\text{pack}}$, we developed the $\Pi_{\text{2lev}}$ construction (see Figure 9) which uses different database patterns to speed-up Setup, can be configured to run Setup efficiently on platforms with a wide range of internal memory, and supports much faster retrieval as a result of higher goodput.

Recall that in $\Pi_{\text{2lev}}$, the EDB consists of two data structures: a dictionary $\gamma$ and an array $A$. The dictionary is again implemented as a bucket hash, but now with exactly one labeled location per keyword $w$. The bucket address and location label are derived from $w$, but the location within the bucket is selected at random to ensure history independence. A $\gamma$ location stores up to $b$ tuples or $b'$ pointers, i.e. indices in array $A$.

The second data structure is the array $A$ whose entries are called *tuple blocks*. Setup uses *tuple blocks* to store tuples, or tuples and pointers, for medium or large $\mathsf{DB}(w)$, respectively. Each *tuple block* stores up to $B$ tuples or $B'$ pointers, with $B \gg b$ and $B' \gg b'$ in most settings. In contrast to the dictionary $\gamma$, which is a bucket hash, the array $A$ needs not be over-sized except for the purpose of masking the total number of tuples in EDB. Empty locations in $\gamma$ and $A$, if any, are filled with random bits.

For all $w$ with more than $|\mathsf{DB}(w)| > b$, the *tuple block*s used for $\mathsf{DB}(w)$ are allocated at random in the array using an AES-based pseudorandom permutation and the tuple list in $\mathsf{DB}(w)$ is split into *tuple block*s

[1] Stripe is the smallest amount of data that can be addressed within the RAID. This is functionally equivalent to a block for an individual disk.

(see medium/large cases in Figure 9). For any given $w$, if the number of *tuple block*s needed to store $\mathsf{DB}(w)$ is larger than the number of pointers that fit in a dictionary location, we use additional *tuple block*s to store pointers (see large case Figure 9).

The dictionary $\gamma$ and the array $A$ are realized as two separate files on the same or separate ext4 RAID partitions. The location, bucket and *tuple block* sizes are configurable, but for efficiency the bucket and *tuple block* sizes must be a multiple of the RAID stripe size. Similarly, buckets and *tuple block*s must be aligned with the pages of the underlying file system.

In our experiments, we use a low single digit number of tuples per location and 32KB or 64KB for buckets and *tuple block*s. Pointers are 3 or 4 bytes long, depending on the size of the array $A$, and tuples are between 16 and 91 bytes long, depending on the functional setting. For the document collections and relational databases in our experiments, the dictionary is between one and two orders of magnitude smaller than the array.

Unpadded, the size of the dictionary leaks the approximate number of keywords while the size of the array leaks the approximate number of EDB tuples. Therefore, masking the dictionary size, which is sensitive in many scenarios, is inexpensive given its relative small size. Leaking the total number of tuples is less sensitive, which means that the larger data structure requires less or no padding in most common cases.

This construction has several important advantages for very large datasets, in particular for those having multi-modal distributions, e.g., some $\mathsf{DB}(w)$ sets that are very large and a very large number of very small $\mathsf{DB}(w)$ as commonly encountered. For instance, for datasets of tens of millions of documents, each English word that is not rare can be found in millions or more documents. On the other hand, ISBN or SSN values are associated with only one book or person, respectively, independent of how many books or persons the dataset describes.

$\Pi_{\text{2lev}}$ can be configured to be disk-efficient in both extremes. For rare keywords, configurations with small location sizes, corresponding to a low single digit number of tuples, allow the search engine to retrieve all the relevant tuples with only *one* disk access. Using a small location size helps reduce the dictionary size, potentially to less than the amount of the available RAM.

For the rest of the keywords, after *one* access (or a few disk accesses for very common keywords), the addresses of all the relevant *tuple block*s are known. At this point, the query execution engine issues as many concurrent *tuple block* requests as the RAID controller can handle. After less than the average disk access time, because of the large number of pending requests, *tuple block*s are read at close to the maximum rate of the RAID controller. The rate at which tuples are retrieved from storage determines the throughput of search engine. Note that goodput is 100% when accessing *tuple block*s filled with tuples and that for frequent keywords, the goodput of a Search operation grows asymptotically to 100%.

In contrast, and by way of comparison, the $\Pi_{\text{pack}}$ construction computes the location addresses of all their tuple groups from the keyword value and a running counter. Thus it can precompute a large number of group addresses and issue requests for tuple groups immediately, i.e. no additional disk accesses to retrieve pointers are needed. But without a priori knowledge of $\mathsf{DB}(w)$ size, which is the common case, $\Pi_{\text{pack}}$ issues many more requests than necessary. Even worse, to achieve the lowest access latency for a CH-based construction, one always needs to issue two requests per expected tuple group, as the group can be stored in two positions (pages) in the CH table. Finally, these disk accesses have low goodput as each bucket contains multiple tuple groups. Thus it appears that low I/O goodput is inherent to $\Pi_{\text{pack}}$. For large sets, the superior goodput of our construction (due to large *tuple block*s) more than compensates for the extra initial storage access(es) for pointers.

For keywords with just a few tuples that fit in one dictionary location, the performance is the same. However, one could accelerate the performance of $\Pi_{\text{2lev}}$ by storing the dictionary, which is relatively small even for large data sets, in main memory. Dictionaries used by previous work, which use one large bucket hash for all tuple sets, are too large for this optimization.

The two-level $\Pi_{\text{2lev}}$ construction allows for a *very efficient* EDB *generation* process. As an example, during the longest phase of EDB generation from a database with $\approx 25$ billion $(w, \mathsf{id})$ pairs in the context of multi client OXT [4], which took 40 hours, all cores performed crypto operations at close to 100% utilization while at the same time reading 100 million records from a MySQL DB and writing to the file system the *tuple* *block*s and the temp dictionary files. Overall, the two-level construction is much closer to our requirements than any previous ones and the experimental results confirm our expectations.

## 5.3 EDB Generation

For our largest datasets, EDB is on the order of 2TB. Thus EDB generation time is sensitive to implementation details and is the dominant factor determining the practical scalability of all our constructions. This section describes the parallel Setup algorithm used in the $\Pi_{2\text{lev}}$ prototype.

Before EDB generation starts we process the input files into an index of the form expected by $\Pi_{2\text{lev}}$. For each text column 't' in the clear-text database table create a new 'text_t' table with columns *ind* and *word*. For each clear-text record and for each word 'xxxx' in its text column, add the pair $(\mathsf{id}, \text{xxxx})$ to 'text_t', where id is the unique identifier of the clear-text record. The number of pairs in 'text_t' is equal to the number of clear-text records multiplied by the average number of words in column 't'. At the end, we create an index on the column 'word', which is used during Setup to retrieve $\mathsf{DB}(w)$ for all $w = (t, \text{xxxx})$, where 'xxxx' is a word in column 't'.

Unfortunately, for our largest databases, 'table_t' is too large for the index to fit in RAM, which makes building the index impractical. To overcome this obstacle, for each text column 't' we create multiple tables 'text_t_nn', such that (1) id-word pairs are somewhat evenly distributed across the new tables, (2) all the pairs with the same 'word' value are in the same table, and (3) the new tables are small enough for their indexes to be built efficiently on our platforms. Note that the atomic columns of the original table can undergo a similar transformation if the number of records in the clear-text table is too large for indexing.

EDB is generated in three phases. The first phase counts the number of distinct keywords $w_i$ in the clear-text table and other statistics needed for sizing the dictionary $\gamma$ and array $A$ (or for masking the sizes of these data structures if so desired). This phase uses full-index scans and takes a small fraction of the total EDB generation time.

For performance reasons, the dictionary $\gamma$, realized as a bucket hash, is partitioned in equal size groups of consecutive buckets and its generation is split across the second and third phases. The *tuple block* array $A$ is fully generated in the next phase.

The second phase generates the tuples in $\mathsf{DB}(w)$, for all keywords $w = (i, val)$ using full-index scans on atomic column $i$. For each text column 't', the newly created 'text_t_nn' tables are scanned. Columns are processed in parallel worker threads, with approximately 1.5 workers per CPU core to hide database access latencies. For each value $val$ such that $w = (i, val)$, the thread processing column $i$ retrieves the all the ids corresponding to the records with $val$ in column $i$ and applies a random permutation to the resultant id sequence (i.e., $\mathsf{DB}(w)$). For each id in the permuted sequence, the worker generates tuple elements with the encrypted id (and the additional tuple values rdk and $y$ when implementing the more advanced features of the OXT protocol from [4]).

During this phase, each worker thread creates one temp file per dictionary partition in which it stores the content of the locations (tuples or pointers) assigned to any of the buckets in the partition. For better performance, the location content is buffered and appended to the partition file in 64KB data chunks. At the same time, for medium and large $\mathsf{DB}(w)$, the worker threads create all the necessary *tuple block*s in the array $A$ (see Figure 9).

During the third phase, the dictionary $\gamma$ is created from the partition files generated in the previous phase. Each partition is constructed by a separate worker thread. Each worker thread reads the files generated by phase-two workers for its partition, merges their contents and creates the label and content of each dictionary entry in the partition. Next, for each bucket in its partition, the worker assigns the dictionary entries to random locations in the bucket, fills in empty locations with random bits, and writes the bucket to disk. For a typical census table, the dictionary file is almost two orders of magnitude smaller than the tuple block file.

Note that for the creation of the dictionary, the file system is accessed using append calls (to temp files in the second phase) and sequential read calls (from the temp files in the third phase), which are the most efficient ways to access large files.

However, worker threads still issue a large number of random write calls during the second phase, with one call for each *tuple block* generated. To reduce the disk head movements associated with these writes, we extended the parallel EDB generation algorithm to confine concurrent workers to a *tuple block window* sliding across the array. As a result, *tuple block*s that are closely located on the disk are generated near simultaneously. This optimization reduces seek overheads noticeably for our largest EDBs.

During the third phase, threads issue another set of random writes when writing the dictionary buckets. These disk movements generated by these writes do not represent a major bottleneck because these writes are already clustered to the bucket hash partitions under constructions, which we always select in increasing order.

## 5.4 Complex Functional Settings

As already mentioned at the end of Section 2, our constructs can be used to support richer encrypted-search settings than SSE, such as those in [4, 14]. In particular, all (single-keyword) SSE schemes presented here can be used, almost 'out-of-the-box', to instantiate the "TSet functionality" underlying the OXT protocol in the above works. The main change is on the size of tuples that increases in order to accommodate additional OXT information such as the xind and $y$ values (see Section 3.2 of [4]), and the key to decrypt the actual documents (as required in multi-client settings [14]).

Storing larger tuples requires minor configuration changes but no alteration of the original construct. More specifically, hash table buckets need to be made large enough to accommodate enough entries for all the tuples to be inserted into the table with high enough probability, i.e., without any of the buckets overflowing.

Another challenge in more complex protocols, such as OXT, is for the server to efficiently perform a two party computation which takes in-order generated data by the client and out-of-order the tuples, as retrieved from the disk by the $\Pi_{\text{pack}}$ or $\Pi_{\text{2lev}}$ prototypes. Maximizing the throughput of such a computation requires using complex buffer management algorithms that optimize the use of available RAM between tokens and *tuple block* buffers.

## 5.5 Experimental Results

Our prototype implementation measures roughly 65k lines of C code, including test programs. Measurements reported here were performed on blades with dual Intel Xeon 2.4GHz E5620 processors having 4 cores each and running Linux. Storage consists of 6 1TB SAS disks configured as a RAID-0 with a 64KB stripe and attached via a 3 Gb/s to an LSI 1064e SAN controller and formatted as an ext4 file system with an 8KB page size. Clear-text databases are stored in MySQL version 5.5.

The experiments reported in this section use databases derived from the ClueWeb Collection [19] or synthetically generated by an engine trained on US-census data. The key attributes of these databases and derived encrypted indices are summarized in Table 1. The `CW-*` databases were extracted from the ClueWeb Collection while the `LL-*` databases emulate the US-census data. Both database families contain atomic type and text columns. The ClueWeb databases were encrypted for a multi-client setting supporting conjunctions (OXT) [4] and the census database where processed for single keyword search (SKS), also in multi-client settings [14] (see Section 5.4).

As already mentioned, EDB generation is the dominant factor determining the practical scalability of all our constructions. The two plots called `CW (PH)` and `CW (2L)` in Figure 10 show how long it takes to generate the EDBs corresponding to the four `CW-*` databases when using the $\Pi_{\text{pack}}$ and $\Pi_{\text{2lev}}$ prototypes, respectively.

The results clearly show the $\Pi_{\text{2lev}}$ construction outperforming the $\Pi_{\text{pack}}$ one. The $\Pi_{\text{pack}}$ prototype is consistently slower because its database access patterns are more expensive than those of the $\Pi_{\text{2lev}}$ prototype. For larger datasets, the performance of the $\Pi_{\text{pack}}$ prototype collapses as soon as its RAM requirements, which are proportional with the database size, approach the available RAM. The $\Pi_{\text{2lev}}$ prototype does not exhibit a similar pattern because its RAM requirements are roughly proportional with the size of the database columns currently being processed.

| DB Name | Records | $(w, \mathsf{id})$ pairs | EDB size |
|---|---|---|---|
| CW-MC-OXT-1 | 408,450 | 143,488,496 | 69.6 GB |
| CW-MC-OXT-2 | 1,001,695 | 316,560,959 | 99.8 GB |
| CW-MC-OXT-3 | 3,362,993 | 1,084,855,372 | 242.4 GB |
| CW-MC-OXT-4 | 13,284,801 | 2,732,311,945 | 903.9 GB |
| LL-MC-SKS-1 | 100,000 | 114,482,724 | 15.0 GB |
| LL-MC-SKS-2 | 1,000,000 | 1,145,547,173 | 52.0 GB |
| LL-MC-SKS-3 | 10,000,000 | 11,465,515,716 | 394.0 GB |
| LL-MC-SKS-4 | 100,000,000 | 114,633,641,708 | 3,961.3 GB |

Table 1: Databases

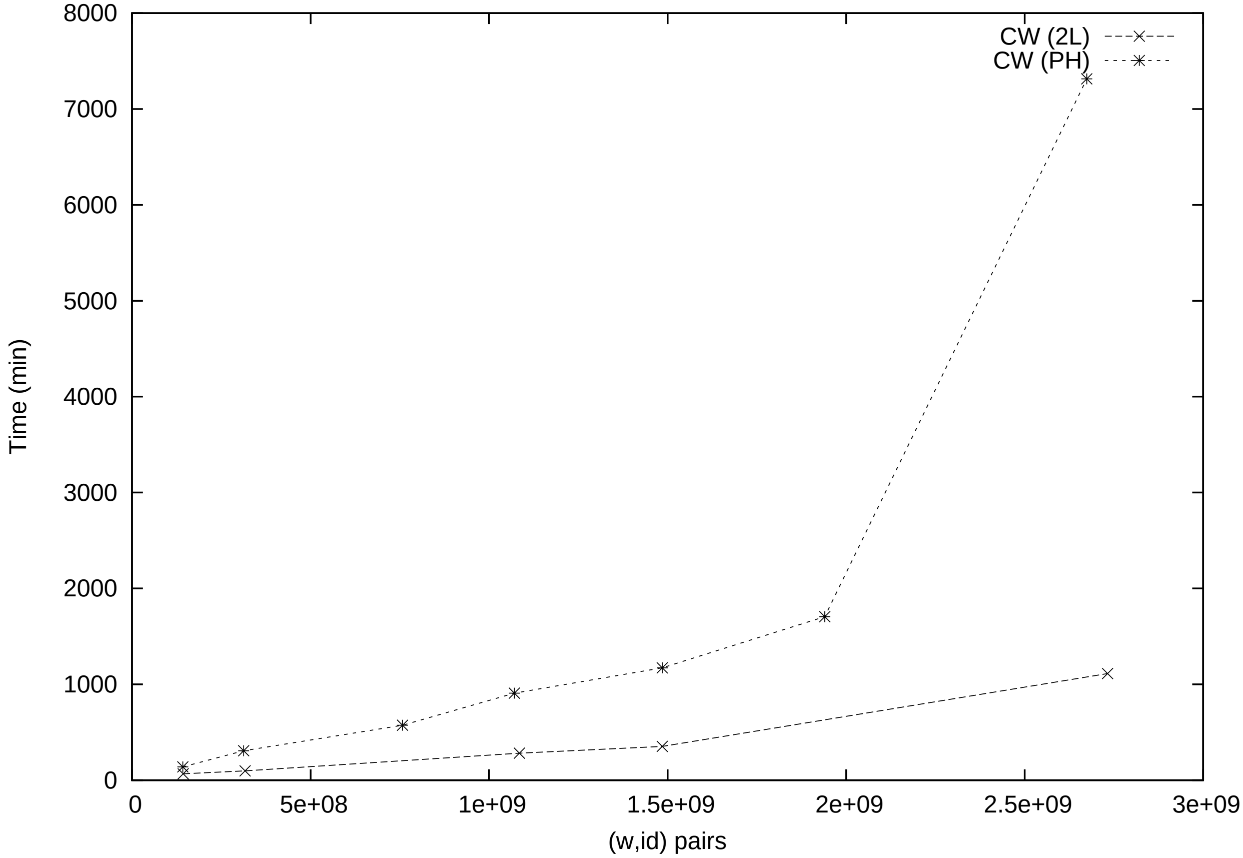

Figure 10: ClueWeb09 Pre-processing: Scaling Database Size

In separate experiments with the $\Pi_{2\text{lev}}$ prototype, preprocessing for the `LL-*` family of databases also proved to scale linearly, with roughly a rate of $3\mu$s per $(w, \mathsf{id})$ pair for the largest database and $8.9\mu$s per $(w, \mathsf{id})$ pair for the smallest one. This translates to roughly 92 hours for biggest `LL-MC-SKS-4` database as shown in Figure 11. This compares very favorably to the experimental results published by Kamara et al [16], who report a cost of approximately $35\mu$s per $(w, \mathsf{id})$ pair on a computing platform with similar capabilities.

Measurements on query performance are shown in Figure 12 for queries `CW-*` databases with varying result set sizes and for both constructions. The graph demonstrates even more dramatic improvements for query processing compared to pre-processing as the $\Pi_{2\text{lev}}$ construction outperforms the $\Pi_{\text{pack}}$ one by almost two orders of magnitude for queries returning 1% of the database.Experiments with the $\Pi_{\text{pack}}$ prototype returning

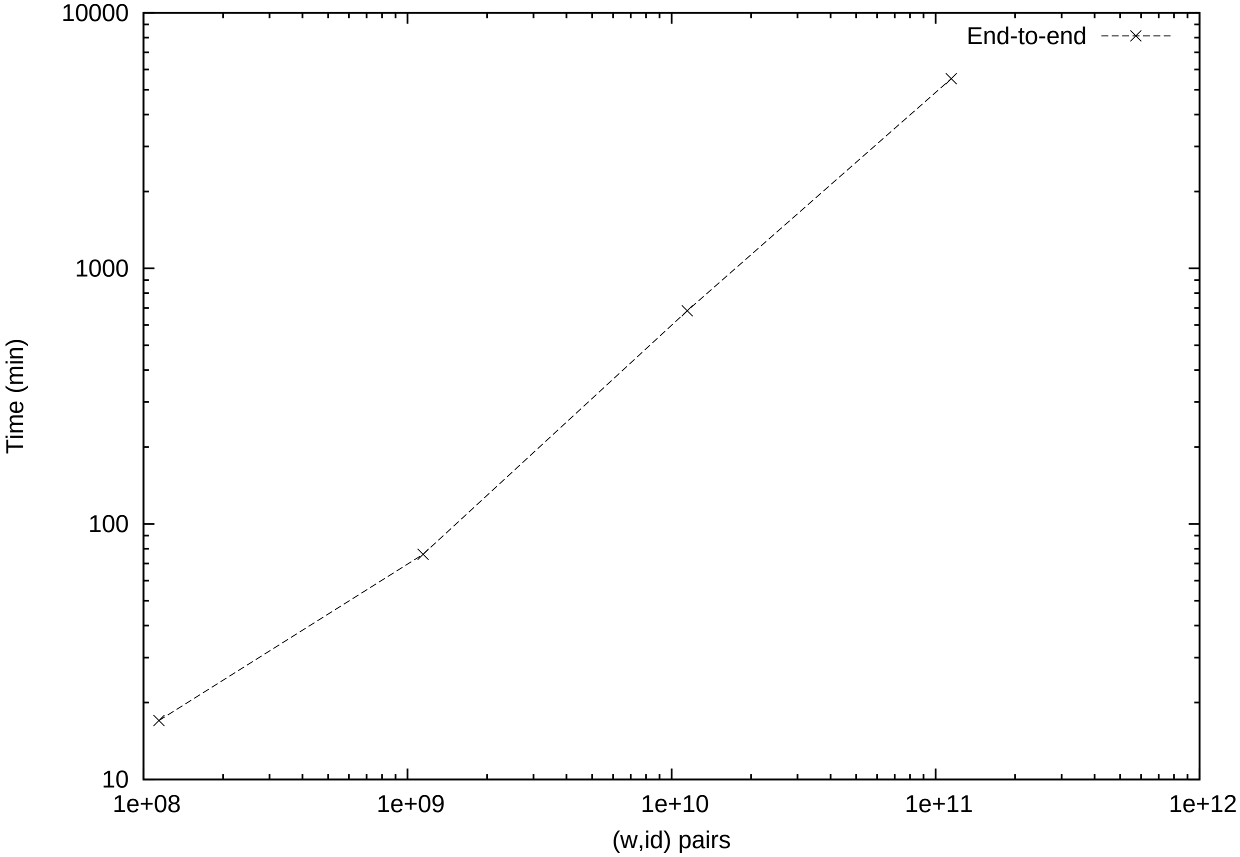

Figure 11: LL SKS Pre-processing: Scaling Database Size

13% of the database are much slower and they were not included in the figure to improve the visibility of the existing curves. Experiments with OXT demonstrate similar performance gains on conjunction queries. This illustrates that even though OXT performance seemingly is dominated by exponentiation costs (see [4] for the details), optimizing disk-resident data-structures are crucial for good performance due to the high I/O latency costs.[2]

Figure 13 shows the execution times of two queries returning a constant, i.e., independent of the size of the input dataset, result set of 10 and 10,000 record ids, respectively. The gap between the two lines corresponding to the $\Pi_{\text{pack}}$ prototype is much larger than the gap between the lines for corresponding to the $\Pi_{\text{2lev}}$ prototype. The difference between the disk layouts of the two constructs help explain the difference. To retrieve the 10 or the 10,000 ids, the $\Pi_{\text{pack}}$ prototype needs to access one or one thousand hash table buckets, respectively, which means it will issue one thousand times more disk access requests for the second query. In contrast, for the same two queries, the $\Pi_{\text{2lev}}$ prototype needs to access one dictionary entry in both cases and one or eleven *tuple block*s, which means it will issue only six times more disk access requests for the second query. $\Pi_{\text{pack}}$ hash table buckets and the $\Pi_{\text{2lev}}$ dictionary buckets and *tuple block*s are all 64KB but tuple groups store only ten tuples in this $\Pi_{\text{pack}}$ prototype while *tuple block*s store a little under one thousand tuples. Note that since $\Pi_{\text{pack}}$ and $\Pi_{\text{2lev}}$ experiments are using SKS and OXT, respectively, the gap between

[2] Using highly optimized (common-base) NIST P-224 exponentiation and multi-threading, we can achieve almost 500,000 exponentiation/sec on the mentioned test bed. The storage system provided only, depending on block size, 500-1,500 random I/O requests/sec and single request latencies is around 10ms.

the 2L and PH plots for experiments returning 10 tuples is explained by the initial computational overhead of OXT.

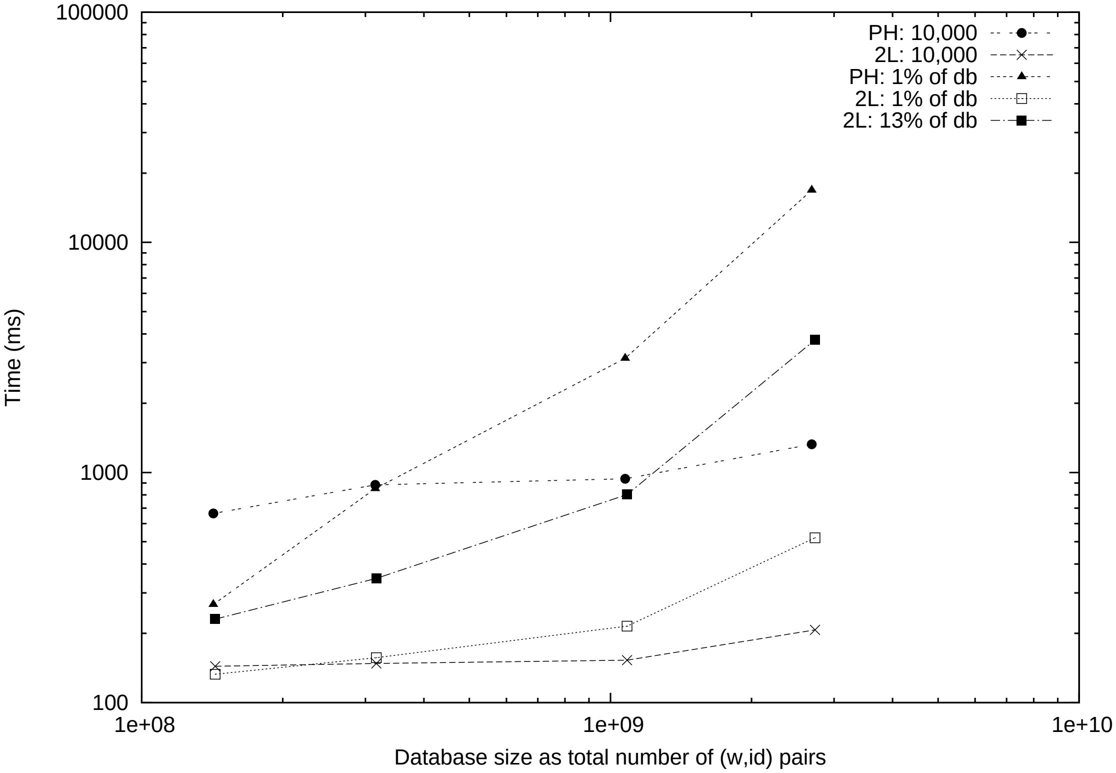

Figure 12: Clueweb09 SKS Query: Scaling Database Size, comparing $\Pi_{\text{pack}}$ vs $\Pi_{\text{2lev}}$ for various result set sizes.

Lastly, to illustrate how space efficient the $\Pi_{\text{2lev}}$ construction is, we achieve load-factors of 58.5% for the bucket hash dictionary, 91.9% for the much larger array and 91.6% overall for our largest LL-MC-SKS-4 database. The load-factor of the array $A$ is less than 100% because although all its entries are used for *tuple blocks*, some of these *tuple blocks* store pointers or are only partially filled with tuples.

## 5.6 Comparison with Prior Implementations

The only previous work that stores the encrypted index on external storage is [4], which uses a construction similar to SSE-2 in [8] but adapted to external storage. It corresponds, roughly, to our $\Pi_{\text{pack}}$ prototype discussed in Section 5.1. The other previous works assume that the encrypted index is stored in main memory and that access to the index data structure is uniform (i.e., constant cost/latency). None of these constructions admit straightforward extensions to 'block device'-resident data structures that can be practical. This is particularly the case for constructions using linked lists, such as SSE-1 in [8] or its extension to dynamic SSE in [16].

Recent work by Kamara et. al in [15] discusses an extension of their main memory index to a storage-resident B-tree. This system suffers from using a large index (worst-case quadratic) and their achieved CPU parallelism does not automatically translate to efficient I/O parallelism given the different characteristics of storage sub-systems. The work of [15] does not measure implementation performance and it does not discuss how it would address the I/O challenges faced when building a large scalable system. In contrast, we identify parallel I/O access from the outset as one of the most important performance requirements for scaling to large encrypted indexes. In addition, we also achieve excellent CPU parallelism during search because we

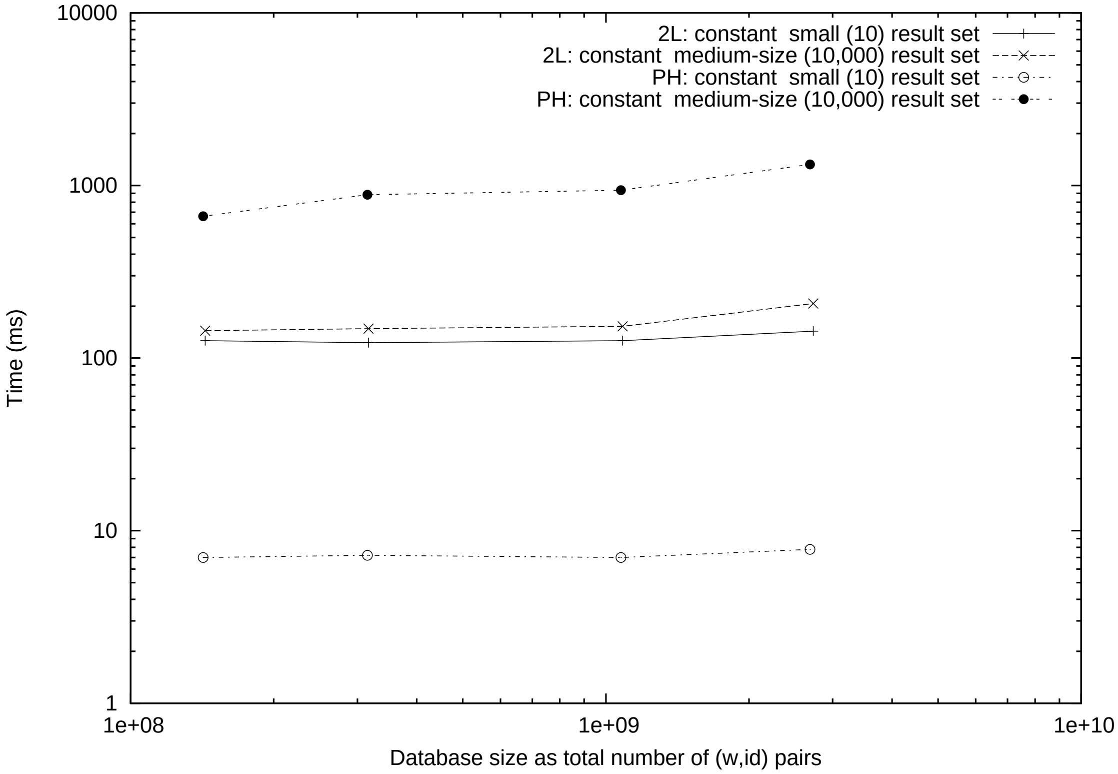

Figure 13: Clueweb09 SKS Query: Scaling Database Size, comparing $\Pi_{\text{pack}}$ vs $\Pi_{\text{2lev}}$ for constant (10 and 10,000) result set sizes.

parallelize our application-level implementation and because a large fraction of the I/O code path is run in parallel kernel threads. We also validate our approach with experimental results, paramount given the intricacies of storage sub-systems. Finally, our construction does not require a fixed keyword set and is asymptotically faster by $\log n$ than the tree construction in [15], as we use hash instead of tree indexing.

## 6 Conclusions

The tension between security and performance requirements for SSE systems pose non-trivial challenges for both the cryptographic design and the data structures needed to support this design, as well as for their implementation. Leakage minimization calls for randomization of data locations in the encrypted database, EDB, in order to obscure any relations in the original clear-text database. This results in the need to randomize access to EDB elements even when these elements are logically correlated (e.g., the set of documents containing a given keyword). This random-order access is affordable for RAM-resident EDB but becomes prohibitive for disk-resident ones; on the other hand, restricting an implementation to a RAM-resident EDB means limiting the database sizes one can support. Thus, much of the work reported here, both at the abstract data structure level and the specifics of their implementation, are driven by the need to bridge over this security-performance conundrum, and is intended to find a balance between randomized ordering of data, locality of access and parallel processing. In particular, our two-level scheme seems to achieve a desirable trade-off between these competing requirements.

As a result we are able to demonstrate the practicality of search on encrypted data for very large datasets (at terabytes-scale and 10s of billions of record-keyword pairs) and with strong security guarantees. Moreover, our implementation experience shows that even complex queries, as those supported in the work of [4], that go well beyond the single-keyword search capability of traditional SSE schemes, can be supported in practical ways for these very large databases. The same is true for the complex operational settings of [14] that support delegation and authorization of queries to multiple clients as well as providing query privacy from the data owner.

Regarding the security of our schemes, it is important to highlight that while the leakage of our constructions compares well with related work, there is non-trivial information being disclosed to the server about result sets (e.g., the size of these sets and their intersections). When the server has a priori information on the database and queries, various statistical attacks are plausible as illustrated, for instance, in [13]. To mitigate such attacks one can apply various generic masking counter-measures such as padding the sets $\mathsf{DB}(w)$ with dummy elements or batching multiple updates to obfuscate update patterns. Hopefully, future work will shed more light on the best ways to design such masking techniques. In particular, one confronts the hard problem of how the syntactically-defined leakage can be captured in a semantic way such that for real world data sets and query distributions one can decide how much and what type of masking approaches are effective.

As mentioned in the introduction, an attractive alternative to achieve more secure solutions to the SSE problem is the use of Oblivious RAM (ORAM) for which we have seen tremendous progress recently in terms of practical performance. However, nobody to our knowledge has yet systematically assessed on how to implement leakage-free search algorithms on top of ORAM servers. Even if we would tolerate the amount of leakage equivalent to our constructions, it is not clear whether one could achieve a similar level of performance for ORAM when considering critical practical aspects such as parallelism and interleaving of I/O and computation as exploited in our approach. Furthermore, the extensibility of ORAM-based solutions to scenarios such as multi-client poses even further challenges.

## Acknowledgment

Supported by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior National Business Center (DoI / NBC) contract number D11PC20201. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, DoI/NBC, or the U.S. Government.

## References

[1] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73, Fairfax, Virginia, USA, Nov. 3–5, 1993. ACM Press. 4

[2] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Berlin, Germany. 4

[3] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In C. Cachin and J. Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 506–522, Interlaken, Switzerland, May 2–6, 2004. Springer, Berlin, Germany. 4

[4] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 353–373, Santa Barbara, CA, USA, Aug. 18–22, 2013. Springer, Berlin, Germany. 1, 2, 3, 4, 8, 22, 23, 24, 25, 26, 28, 29, 31

[5] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In J. Ioannidis, A. Keromytis, and M. Yung, editors, *ACNS 05*, volume 3531 of *LNCS*, pages 442–455, New York, NY, USA, June 7–10, 2005. Springer, Berlin, Germany. 1

[6] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 577–594, Singapore, Dec. 5–9, 2010. Springer, Berlin, Germany. 1, 3, 4, 6, 13

[7] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, 1998. 4

[8] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In A. Juels, R. N. Wright, and S. Vimercati, editors, *ACM CCS 06*, pages 79–88, Alexandria, Virginia, USA, Oct. 30 – Nov. 3, 2006. ACM Press. 1, 3, 5, 6, 29

[9] M. Dietzfelbinger, M. Mitzenmacher, and M. Rink. Cuckoo hashing with pages. Technical Report abs/1104.5111, arXiv, 2011. 23

[10] E.-J. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. `http://eprint.iacr.org/`. 1

[11] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. 4

[12] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *CCSW*, pages 95–100, 2011. 4

[13] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2012)*, San Diego, CA, Feb. 2012. Internet Society. 31

[14] S. Jarecki, C. Jutla, H. Krawczyk, M. C. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *ACM CCS 13*, Berlin, Germany, Nov. 4–8, 2013. ACM Press. 3, 8, 21, 26, 31

[15] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In A.-R. Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 258–274, Okinawa, Japan, Apr. 1–5, 2013. Springer, Berlin, Germany. 1, 3, 21, 29, 30

[16] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM CCS 12*, pages 965–976, Raleigh, NC, USA, Oct. 16–18, 2012. ACM Press. 1, 3, 6, 21, 27, 29

[17] J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007. 4

[18] K. Kurosawa and Y. Ohtaki. UC-secure searchable symmetric encryption. In A. D. Keromytis, editor, *FC 2012*, volume 7397 of *LNCS*, pages 285–298, Kralendijk, Bonaire, Feb. 27 – Mar. 2, 2012. Springer, Berlin, Germany. 1, 3

[19] Lemur Project. ClueWeb09 dataset. `http://lemurproject.org/clueweb09.php/`. 26

[20] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *2000 IEEE Symposium on Security and Privacy*, pages 44–55, Oakland, California, USA, May 2000. IEEE Computer Society Press. 1, 4

[21] E. Stefanov, E. Shi, and D. X. Song. Towards practical oblivious RAM. In *NDSS 2012*, San Diego, California, USA, Feb. 5–8, 2012. The Internet Society. 4

[22] P. van Liesdonk, S. Sedhi, J. Doumen, P. H. Hartel, and W. Jonker. Computationally efficient searchable symmetric encryption. In *Proc. Workshop on Secure Data Management (SDM)*, pages 87–100, 2010. 1, 3, 21