IO-DSSE: Scaling Dynamic Searchable Encryption to Millions of Indexes By Improving Locality

Ian Miers [*] Payman Mohassel [†]

August 26, 2016

Abstract

Free cloud-based services are powerful candidates for deploying ubiquitous encryption for messaging. In the case of email and increasingly chat, users expect the ability to store and search their messages persistently. Using data from one of the top three mail providers, we confirm that for a searchable encryption scheme to scale to millions of users, it should be highly IO-efficient (locality), and handle a very dynamic message corpi. We observe that existing solutions fail to achieve both properties simultaneously. We then design, build, and evaluate a provably secure Dynamic Searchable Symmetric Encryption (DSSE) scheme with significant reduction in IO cost compared to preceding works when used for email or other highly dynamic message corpi.

1 Introduction

The last few years have seen incredible success in deploying seamless end-to-end encryption for messaging. Between iMessage, WhatsApp, and SRTP for video, over 2 billion users routinely encrypt their messages without necessarily even noticing. Unfortunately, this trend has been largely limited to ephemeral or semi-ephemeral communication mediums. Where users expect messages to be available for later recall, we have seen little progress.

The quintessential example of this is email, where messages are searched for months or years after being received. But email is not the only medium filling this role: Google Hangouts, Slack, and Facebook messenger all operate in an archive and search paradigm—indeed the latter two are to some extent used as an email substitute because of this. In this setting, end-to-end encryption is not seamless: by using it, users lose the ability to store messages in the cloud and search. Since users are unwilling to sacrifice features for security, this is a major impediment to deploying end-to-end encryption.

Symmetric Searchable Encryption (SSE) [6, 7, 8, 10, 15, 19] provides a potential solution to this problem as it allows a client to outsource an encrypted index of documents (e.g. emails) to an untrusted server and efficiently search the index for specific keywords. The efficiency of SSE stems from its use of fast symmetric-key operations and its privacy guarantees typically allow for some information leakage on search/access patterns which is captured using a leakage function. The dynamic variants of SSE (DSSE) [5, 13, 14, 20] also allow for efficient updates to the encrypted database. The question is wether dynamic SSE can be used efficiently in a cloud-scale application?

Several works [5, 6] have addressed scaling SSE schemes to a very large index— terabytes of data— but to the best of our knowledge, none have examined deploying millions of small indexes on

[*] Johns Hopkins University, imiers@cs.jhu.edu.

[†] Visa Research, pmohasse@visa.com. Work done while at Yahoo Labs.

	IO to insert n documents with k keywords		IO for a search returning n documents		supports deletes
index type	Static	Dynamic	Static	Dynamic	
Unencrypted Index	$O(k)$	$O(k)$	$O(1)$	$O(1)$	yes
Cash et. al [5]	$O(k)$	$O(nk)$	$O(1)$	$O(nk)$	dynamic only
Stefanov et. al [20]	$O(nk)$	$O(nk)$	$O(nk)$	$O(nk)$	yes
This work[1]	$O(k)$	$O(k)$	$O(1)$	$O(1)$	partial dynamic

Table 1: Informal description of IO costs of SSE schemes assuming n, the number of documents mapped to a given keyword, is smaller that the block size used for storage. When inserting n documents containing k keywords into an existing index, all existing SSE schemes write each posting (i.e. keyword, document ID pair) to a distinct random location. As a result, when adding n documents to the index entry for a given keyword, there are n writes to distinct random locations. This is repeated for each of the k keywords. For search, returning the complete entry in the index for a given keyword requires each of the n random locations be read. When used for messaging or email, all insertions and searches are done against the dynamic index and any savings in the static case do not apply.

shared hardware. This is the exact problem faced by a cloud provider wishing to deploy search for encrypted messaging or email with deploying large amounts of new hardware.

Since SSE schemes typically use fast symmetric cryptography, the primary performance issue for SSE is IO and specifically poor locality: unlike a standard inverted index where the list of document ID's for a given keyword k can be stored in a single location, SSE schemes typically place each document ID for a given keyword in a distinct random location in order to hide the index structure. This results in a large increase in IO usage, since searching for a keyword found in e.g. 500 documents results in 500 random reads, rather one single read retrieving a list of 500 document IDs. Similarly, inserting a batch of e.g. 100 documents containing some 150 total keywords, requires $100 \times 150 = 15,000$ writes, rather than one write to update each.

While the same locality issues arise in SSE schemes for very large indexes, the solution space is far more constrained in our setting and the main techniques introduced in the most promising approaches [5, 6] are not applicable for purely dynamic indexes (i.e. indexes that are initially empty). As a result, deploying existing SSE schemes for search for mail or messaging—where ever entry is inserted dynamically into an initially empty index—-would result in an order of magnitude increase in IO usage. Since existing mail search is already IO bound, the cost of doing so is prohibitive. This is a marked departure from the cost of deploying end-to-end encryption for ephemeral communication, which comes at little operational cost.

Unfortunately, this increased cost cannot be handled by simply using better hardware. First high performance storage systems are prohibitively expensive relative to profit from free communications services. Second caching is relatively ineffective as each individual index is used infrequently. Third, Even given cost effective storage (e.g. much cheaper SSDs) and the willingness of providers to upgrade infrastructure, any infrastructure-related improvement yields similar cost savings for non-encrypted search. Existing SSE schemes simply require considerably resources than unencrypted search.

We now examine in detail the specific challenges for email and then detail our proposed solution.

[1] We assume the obliviously updatable index is of fixed size and thus the overhead it imposes is constant. This is the case if the key word list is fixed or the server has limited space. Otherwise there is a cost that is logarithmic in the size of the OUI, but independent of total index size or search frequency.

1.1 Why Email is Different

Cloud-based mail offers a unique setting. First, while storage at this scale is cheap, access to it is not: existing mail search is already IO bound. As a result, we must *minimize the IO cost* of maintaining the index. As we will see, this is the driving design constraint behind our work.

Second, unlike traditional settings for SSE, there is no initial corpus of documents to index. Instead all documents (email messages) are added after database creation via updates[2]—the typical user receives between 30 and 200 messages a day.[3] This exacerbates the IO problem, since all existing schemes end up performing one random write to the disk per keyword for each new email and one random read per document ID returned in a search. This is in marked contrast to a standard index wherein multiple entries are packed into one block and read/written in one shot, resulting in efficient IO usage.

Third, the query rate is exceedingly low. At a major mail provider we see on the order of 250 searches per second *total across all users* on mail content from an order of a few hundred million monthly active users. Searches on public metadata such as sender, date sent, "has attachment", etc. are far more frequent, but searches on mail content, i.e., what would be stored in an encrypted index, are surprisingly rare. Many schemes (e.g. [20]) assume hundreds of queries per second, where it is economical to load the index into memory. At an average 2 queries per inbox per *month*, storing the index in memory is neither cost effective nor remotely practical given the sheer number of users each needing their own separate index.

If the IO problems are resolved, however, the prospects for encrypted mail search are actually fairly good: search for email is in many ways easier than what is typically asked of encrypted search. We analyze detailed usage numbers furnished by a major webmail provider. We conclude that searchable encryption for cloud-based email is surprisingly plausible in terms of functionality if cost is disregarded: the size of the datasets are manageable, search is not frequently used, and most queries are exceedingly simple. In particular, the combination of single keyword plus intersection on public metadata (date, sender, "has attachment"), seems to cover the vast majority of searches. While conjunctive queries would of course be an improvement, they are not strictly necessary. Moreover the availability of cheap storage coupled with the fact that search is infrequent, means the cost of index entries for deleted files is relatively small. This allows us to eschew many of the complexities of fully dynamic searchable encryption because we can afford to delete entries on a best-effort basis. The scalability issues become even easier when one considers that end-to-end encrypted mail is likely personal and the majority of email is not. As a result, only a small fraction of mail will need to be encrypted or indexed using SSE.

Why existing schemes fail Encrypted databases such as Blind Seer [16], Cryptdb [17] and others aim to provide complex private queries as a privacy-preserving replacement to traditional large database applications. Just as those database clusters are unsuited for mail search, so too are their privacy-preserving counterparts.

The DSSE scheme of Stefanov et al. [20] provides impressive privacy protections including efficient deletion of entries, but as each individual index entry (i.e. a mapping for a word to a single document) is stored in a random location, the IO expansion relative to standard search is very large. This is not scalable in the context of deploying hundreds of millions of indexes on a few thousand servers.

This problem is not isolated to Stefanov et al.'s scheme. Indeed, Cash et al. [5] note "One critical source of inefficiency in practice (often ignored in theory) is a complete lack of locality

[2] Since the server knows the contents of existing (unencrypted) email, adding existing unencrypted email to the index is unnecessary and for some schemes ill-advised.

[3] The mean is 30 with a standard deviation of 148.

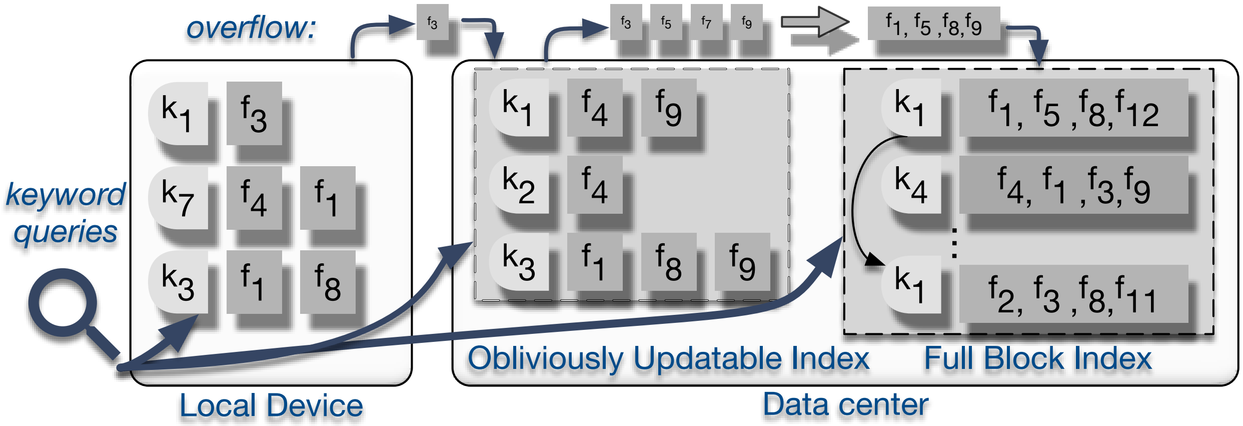

Figure 1: IO-DSSE logical architecture: an index mapping keywords to document IDs is stored locally, then the largest entries overflow into the obliviously updatable index. When the entry for a keyword in the OUI is full, index entries are inserted into the Full Block Index in chunks. Keyword searches query all three indexes. Note: encryption and the obliviously updatable index construction hide this view from the server.

and parallelism: To execute a search, most prior SSE schemes sequentially read each result from storage at a pseudorandom position, and the only known way to avoid this while maintaining privacy involves padding the server index to a prohibitively large size." [5]. Although one can arrive at this from varying perspectives (disk IO, latency, or memory requirements), it is a central problem that renders all previous schemes unusable in this context.

Cash et al. [5] offers a notable improvement on this approach. Instead of storing each index entry at a random location on disk, they store entries sequentially in fixed-size blocks, drastically reducing the number of disk reads by allowing many index entries for a given keyword to be retrieved at once. Unfortunately, this technique cannot readily be used to handle updates. The act of appending a new index entry to an existing (partially-filled) block leaks significant information on updates (which are frequent). This means these techniques are not applicable to entries inserted into the index after the initial index creation.

As a result, while Cash et al. make use of an efficient on-disk index for the starting document corpus, all updates to the index are stored in memory because each index entry is written to a distinct random location. The scheme is not designed to deal with frequent updates in an IO-efficient manner and assumes that most of the corpus is indexed statically, at initialization. Thus, when the scheme is used for dynamic updates, it incurs the very same locality issues the authors identify. Bridging this gap between locality and privacy for highly dynamic indexes is the goal of our construction. See Table 1 for details.

1.2 Our Contribution

A logical extension of Cash et al.'s scheme is to buffer partial blocks locally (client-side) and only upload to the server once the block is full. However our experiments in Section 4.3 indicate that a storage-limited client (e.g. a mobile device) will overflow its local storage in less than 100 days even for the average user. A second approach is to offload this buffer to the server in a way that hides when it is updated but still allows the buffer to be searched. Oblivious RAM (ORAM) [3, 9, 11, 12, 18] is a natural choice. Index entries would be buffered in a small ORAM cache and then written to the full index.

ORAM, however, is not an ideal choice. First, in most SSE schemes, access patterns for search are already leaked, yet in ORAM we must pay for the full ORAM overhead not just for reads and writes associated with index updates, but for searches as well.[4] Second, ORAM must remain secure for arbitrary access patterns, while we only need to obscure batched updates to the index (many emails/keywords can be batched and inserted in the index at the same time).

Thus, in Section 3, we construct an obliviously updatable index (OUI), which provides ORAM-like properties for efficient batched updates but allows simple reads for search. This is accomplished by using a weaker security requirement tailored to the SSE leakage function. Through optimizations and batching, we achieve a 94% IO savings vs. the generic approach using Path ORAM [22].

To summarize our contributions:

1. We examine the requirements and feasibility of search on encrypted email and more generally, the class of IO-bound DSSE schemes that exhibit a low query rate and high update rate. This examination is supported with realistic data from a large webmail provider.

2. Motivated by IO-efficient SSE for highly dynamic corpi, we introduce the concept of an Obliviously Updatable Index (OUI). As a proof of concept, we provide a new construction for an OUI, with proof of security, which offers a 94% savings compared to a naive implementation using ORAM. We obtain our final solution by combining this with the state-of-the-art IO-efficient SSE that indexes the full blocks.

3. We evaluate our solution using real-world data from 100,000 mail users, showing we achieve a 99% reduction in the IO cost vs. the state-of-the-art in SSE schemes such as that of Cash et al. which, for purely dynamic insertions, write each document-keyword pair to a random location. We also report the storage required for a typical mail user both on the client side, and the server side.

1.3 No free lunch

These improvements are, of course, not free. First, we must allow for slightly more leakage in search than the scheme of [5], since we leak when an entry in the OUI is full and needs to be pushed to the full-block index. Second, we can only deal with deletes from the OUI: once an entry is written to the static index, it is stored until the index is rebuilt. Thus we do not provide a fully dynamic index. Given the low cost of storage relative to the size of emails and the fact that for the average user emails are contained fully within the partial index for at least 11 days, we believe this is an acceptable trade-off.

2 Background

2.1 Hash Tables

A hash table is a data structure commonly used for mapping keys to values. It often uses a hash function h that maps a key to an index (or a set of indexes) in a memory array M where the value associated with the key may be found. The keyword is not in the table if it is not in one of those locations. More formally, we define a hash table $H = (\mathsf{hsetup}, \mathsf{hlookup}, \mathsf{hwrite})$ using a tuple of algorithms.

[4] Because we need to know the content of an index entry to update it and must read that from a server, we cannot use so called write-only ORAM which assumes reads are not observable. [2]

- $(h, \mathsf{M}) \leftarrow \mathsf{hsetup}(S)$: hsetup takes as input an initial set S of keyword-value pairs and outputs a hash function h and a memory array M storing the key-value pairs.
- $\mathsf{M}' \leftarrow \mathsf{hwrite}(key, value, \mathsf{M}, h)$: If $(key, value)$ already exists in the table it does nothing, else it stores $(key, value)$ in the table. If M and h are known from the context, we use the shorter notation $\mathsf{hwrite}(key, value)$.
- $value \leftarrow \mathsf{hlookup}(key, \mathsf{M}, h)$: hlookup returns $value$ if $(key, value)$ is in the table. Else it returns $\perp$. If M and h are known from the context, we use the shorter notation $\mathsf{hlookup}(key)$.

2.2 Oblivious RAM

We recall Oblivious RAM (ORAM), a notion introduced and first studied in the seminal paper of Goldreich and Ostrovsky [12]. ORAM can be thought of as a compiler that encodes the memory into a special format such that accesses on the compiled memory do not reveal the underlying access patterns on the original memory. An ORAM scheme consists of protocols ($\textsc{Setup}, \textsc{ObliviousAccess}$).

- $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow \textsc{Setup}\langle (1^\lambda, \mathsf{M}), \perp \rangle$: $\textsc{Setup}$ takes as input the security parameter λ and a memory array M and outputs a secret state σ (for the client), and an encrypted memory EM (for the server).
- $\langle (\mathsf{M}[y], \sigma'), \mathsf{EM}' \rangle \leftrightarrow \textsc{ObliviousAccess}\langle (\sigma, y, v), \mathsf{EM} \rangle$: $\textsc{ObliviousAccess}$ is a protocol between the client and the server, where the client's input is the secret state σ, an index y and a value v which is set to null in case the access is a read operation (not a write). Server's input is the encrypted memory EM. Client's output is $\mathsf{M}[y]$ and an updated secret state σ', and the server's output is an updated encrypted memory EM' where $\mathsf{M}[y] = v$, if $v \neq \mathsf{null}$.

Correctness Consider the following correctness experiment. Adversary A chooses memory M. Consider the encrypted database EM generated with $\textsc{Setup}$ (i.e., $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow \textsc{Setup}\langle (1^\lambda, \mathsf{M}), \perp \rangle$). The adversary then adaptively chooses memory locations to read and write. Denote the adversary's read/write queries by $(y_1, v_1), \dots, (y_q, v_q)$ where $v_i = \mathsf{null}$ for read operations. A wins in the correctness game if $\langle (\mathsf{M}_i[y_i], \sigma_i), \mathsf{EM}' \rangle$ are not the final outputs of the protocol $\textsc{ObliviousAccess}\langle (\sigma_{i-1}, y_i, v_i), \mathsf{EM}_{i-1} \rangle$ for any $1 \leq i \leq q$, where $\mathsf{M}_i, \mathsf{EM}_i, \sigma_i$ are the memory array, the encrypted memory array and the secret state, respectively, after the i-th access operation, and $\textsc{ObliviousAccess}$ is run between an honest client and server. The ORAM scheme is correct if the probability of A in winning the game is negligible in λ.

Security An ORAM scheme is secure if for any adversary A, there exists a simulator S such that the following two distributions are computationally indistinguishable.

- $\text{Real}_{\mathsf{A}}(\lambda)$: A chooses M. The experiment then runs $\langle \sigma, \mathsf{EM} \rangle \leftrightarrow \textsc{Setup}\langle (1^\lambda, \mathsf{M}), \perp \rangle$. A then adaptively makes read/write queries (y_i, v) where $v = \mathsf{null}$ on reads, for which the experiment runs the protocol $\langle (\mathsf{M}[y_i], \sigma_i), \mathsf{EM}_i \rangle \leftrightarrow \textsc{ObliviousAccess}\langle (\sigma_{i-1}, y_i, v), \mathsf{EM}_{i-1} \rangle$. Denote the full transcript of the protocol by t_i. Eventually, the experiment outputs $(\mathsf{EM}, t_1, \dots, t_q)$ where q is the total number of read/write queries.
- $\text{Ideal}_{\mathsf{A},\mathsf{S}}(\lambda)$: The experiment outputs $(\mathsf{EM}, t'_1, \dots, t'_q) \leftrightarrow \mathsf{S}(q, |\mathsf{M}|, 1^\lambda)$.

2.3 Path ORAM

Path ORAM [22] is a tree-based ORAM construction with high practical efficiency. We use Path ORAM as a component in our SSE construction. We only review the non-recursive version of Path ORAM where the client stores the position map locally and hence only a single binary tree T is needed to store the data on the server.

Notations Let M be a memory array of size at most $N = 2^L$ that we want to obliviously store on the server. We use $\mathsf{M}[i]$ to denote the ith block in M. Let T denote a binary tree of depth L on the server side that will be used to store M. The client stores a position map $\mathsf{position}$ where $x = \mathsf{position}[i]$ is the index of a uniformly random leaf in T. The invariant Path ORAM maintains is that $\mathsf{M}[i]$ is stored in a node on the path from the root to leaf x which we denote by $P(x)$. We also use $P(x, \ell)$ to denote the node at level ℓ on path $P(x)$, i.e. the node that has distance ℓ from the root. There is a bucket associated with each node of the tree T, and each bucket can at most fit Z memory blocks.

The client holds a small local stash denoted by S, which contains a set of blocks that need to be pushed into the server's tree.

ORAM Setup We assume that memory array M is initially empty. Client's stash S is empty. All the buckets in the tree T are filled with encryptions of dummy data. The position map $\mathsf{position}$ is initialized with uniformly random values in $\{0, \ldots, 2^L\}$. This encrypted tree is denoted by EM.

Read/Write Operations To read $\mathsf{M}[y]$ or to write a value v at $\mathsf{M}[y]$, the client first looks up the leaf position x from the position map and reads all the buckets along the path $P(x)$. It then updates $\mathsf{position}[y]$ to a fresh random value in $\{0, \ldots, 2^L\}$. If it is a read operation, the encryption of (y, v) will be found in one of the buckets on $P(x)$, which the client decypts to output v. It also adds all the buckets on $P(x)$ to its local stash. If it is a write operation, the client also adds (y, v) to its local stash.

The client encrypts all the blocks in the stash and inserts as many as possible into the buckets along $P(x)$, inserting each block into the lowest bucket in the path possible while maintaining the invariant that each block y' remains on the path $P(\mathsf{position}[y'])$.

Figure 2 describes the read/write operations in more detail. The ReadBucket protocol has the server return the bucket being read to the client who decrypts and outputs the blocks in the bucket. The WriteBucket protocol has the client encrypt and insert all the blocks in its input set into a bucket and send it to the server.

2.4 Searchable Symmetric Encryption

A database D is a collection of documents d_i each of which consist of a set of keywords W_i. A document can be a webpage, an email, or a record in a database, and the keywords can represent the words in the document, or the attributes associated with it. A symmetric searchable encryption (SSE) scheme allows a client to outsource a database to an untrusted server in an encrypted format and have the server perform keyword searches that return a set of documents containing the keyword. For practical reasons, SSE schemes often return a set of identifiers that point to the actual documents. The client can then present these identifiers to retrieve the documents and decrypt them locally.

More precisely, a database is a set of document/keyword-set pair $\mathsf{DB} = (d_i, W_i)_{i=1}^N$. Let $W = \cup_{i=1}^N W_i$ be the universe of keywords. A keyword search query for w should return all d_i where $w \in W_i$. We denote this subset of DB by $\mathsf{DB}(w)$.

```
$\textsc{ObliviousAccess}\langle(\sigma, y, v), \mathsf{EM}\rangle$:
$x \leftarrow \mathsf{position}[y]$
$\mathsf{position}[y] \overset{R}{\leftarrow} \{0, \ldots, 2^L\}$
for $\ell \in \{0, \ldots, L\}$ do
    $S \leftarrow S \cup \textsc{ReadBucket}(P(x, \ell))$
end for
$data \leftarrow$ Read block $y$ from $S$
if $v \neq$ then
    $S \leftarrow (S - \{(y, data)\}) \cup \{(y, v)\}$
end if
for $\ell \in \{L, \ldots, 0\}$ do
    $S' \leftarrow \{(y', data') \in S : P(x, \ell) = P(\mathsf{position}[y'], \ell)\}$
    $S' \leftarrow$ Select $min(|S'|, Z)$ blocks from $S'$.
    $S \leftarrow S - S'$
    $\textsc{WriteBucket}(P(x, \ell), S')$
end for
```

Figure 2: Read/Write Ops in path ORAM

A searchable symmetric encryption scheme Π consists of protocols SSESetup, SSESearch and SSEAdd.

- $\langle \mathsf{EDB}, \sigma\rangle \leftarrow \mathrm{SSESetup}\langle(1^\lambda, \mathsf{DB}), \perp\rangle$: SSESetup takes as client's input a database DB and outputs a secret state σ (for the client) and an encrypted database EDB which is outsourced to the server.

- $\langle(\mathsf{DB}(w), \sigma'), \mathsf{EDB}'\rangle \leftarrow \mathrm{SSESearch}\langle(\sigma, w), \mathsf{EDB}\rangle$:

  SSESearch is a protocol between the client and the server, where the client's input is the secret state σ and the keyword w he is searching for. The server's input is the encrypted database EDB. The client's output is the set of documents containing w, i.e., $\mathsf{DB}(w)$ as well an updated secret state σ', and the server obtains an updated encrypted database EDB'.

- $\langle \sigma', \mathsf{EDB}'\rangle \leftarrow \mathrm{SSEAdd}\langle(\sigma, d), \mathsf{EDB}\rangle$: SSEAdd is a protocol between the client and the server, where the client's input is the secret state σ and a document d to be inserted into the database. The server's input is the encrypted database EDB. The client's output is an updated secret state σ', and the server's output is an updated encrypted database EDB' which now contains the new document d.

Correctness Consider the following correctness experiment. An adversary A chooses a database DB. Consider the encrypted database EDB generated using SSESetup (i.e. $\langle \mathsf{EDB}, K\rangle \leftarrow \mathrm{SSESetup}\langle(1^\lambda, \mathsf{DB}), \perp\rangle$). The adversary then adaptively chooses keywords to search and documents to add to the database. Denote the searched keywords by $w_1, \ldots, w_t$. A wins in the correctness game if $\langle(DB_i(w_i), \sigma_i), \mathsf{EDB}_i\rangle \neq \mathrm{SSESearch}\langle(\sigma_{i-1}, w_i), \mathsf{EDB}_{i-1}\rangle$ for any $1 \leq i \leq t$, where $\mathsf{DB}_i, \mathsf{EDB}_i$ are the database and encrypted database, respectively, after the ith search, and SSESearch and SSEAdd are run between an honest client and server. The SSE scheme is correct if the probability of A in winning the game is negligible in λ.

Security Security of SSE schemes is parametrized by a leakage function L, which explains what the adversary (the server) learns about the database and the search queries, while interacting with a secure SSE scheme.

An SSE Scheme Π is L-secure if for any PPT adversary A, there exists a simulator S such that the following two distributions are computationally indistinguishable.

- $\text{Real}^{\Pi}_{A}(\lambda)$: A chooses DB. The experiment then runs $\langle \mathsf{EDB}, \sigma\rangle \leftarrow \text{SSESetup}\langle(1^{\lambda}, \mathsf{DB}), \perp\rangle$. A then adaptively makes search queries w_i, which the experiment answers by running the protocol $\langle \mathsf{DB}_{i-1}(w_i), \sigma_i\rangle \leftarrow \text{SSESearch}\langle(\sigma_{i-1}, w_i), \mathsf{EDB}_{i-1}\rangle$. Denote the full transcripts of the protocol by t_i. Add queries are handled in a similar way. Eventually, the experiment outputs $(\mathsf{EDB}, t_1, \ldots, t_q)$ where q is the total number of search/add queries made by A.

- $\text{Ideal}^{\Pi}_{A,\mathsf{S},\mathsf{L}}(\lambda)$: A choose DB. The experiment runs $(\mathsf{EDB}', st_0) \leftarrow \mathsf{S}(\mathsf{L}(\mathsf{DB}))$. On any search query w_i from A, the experiment adds (w_i, search) to the history H, and on an add query d_i it adds (d_i, add) to H. It then runs $(t'_i, st_i) \leftarrow \mathsf{S}(st_{i-1}, \mathsf{L}(\mathsf{DB}_{i-1}, H))$. Eventually, the experiment outputs $(\mathsf{EDB}', t_1, \ldots, t'_q)$ where q is the total number of search/add queries made bt A.

3 Our Construction

As discussed earlier, three important security/efficiency requirements for any searchable symmetric scheme for email to be practical are (1) dynamic updatability, (2) low latency on search and hence high IO efficiency, and (3) no leakage on updates (send/received email).

The SSE construction of [6] accommodates dynamic updates and does not leak keyword patterns on updates as each new keyword-document pair is added to the index using a freshly random key generated by a PRF that takes a counter as input. This meets requirements (1) and (3). They also address the IO efficiency issue by storing document IDs in larger blocks and hence retrieving many documents IDs using a single disk access. This provides a partial solution to the IO efficiency and fast search, but the solution is not suitable for a highly-dynamic application such as mail.

In particular, the main challenge is to pack multiple updates (i.e. a set of new keyword-document pairs) into a large block, before pushing them into the encrypted index. The naive solution of storing the partial blocks in the same index and adding new keyword-document pairs on each update leaks the update pattern which is a major drawback for an update-heavy application such as email.

Our construction consists of two pieces: a small dynamic encrypted index for partial blocks, i.e., the **obliviously updatable index** (OUI) and a large append-only encrypted index for full blocks, i.e., **full-blocks index** (FBI).

The full-block index holds a mapping from an encrypted keyword to a fixed-size block containing document IDs. New entries can be added to this index only when we have a full block. This is the basic approach taken by [6] for static encrypted search, though they expand on it to deal with far larger data sets than we wish to.

The main technical challenge in our construction is the design of OUI for managing partially-filled blocks until they are full and can be pushed to the FBI. In particular, note that the blocks in OUI need not be full and are instead padded to some fixed size. When a block is full of real data (i.e. no padding) its contents are transferred to the full-block index. This allows messages to be added and deleted from the OUI by updating the requisite block. Of course, we must do so in a way that does not leak which blocks are being updated, or else we fail to meet the basic requirements for secure search by leaking update patterns.

3.1 An Obliviously Updatable Index

ORAM forms a generic starting point for our obliviously updatable index. In particular, storing partial blocks in ORAM would allow us to update them privately. However, as a generic approach, ORAM is an overkill. An index built on top of ORAM would hide not only reads and writes resulting from an index update, but also reads resulting from an index lookup. This is stronger than the typical protection provided by SSE schemes (which leak the "search pattern"), and in our case (similar to prior work) this information is already revealed via searches against the full-block index. So we gain no additional privacy by hiding the search pattern only in the OUI and will realize considerable efficiency gains by not doing so.

As a concrete starting point, consider a basic construction of Path ORAM [22]. In Path ORAM, entries (called blocks in our construction) are stored as leaves in a full binary tree. Each time an entry is read or written, it is remapped to a random leaf in the tree or, if that position is full, the first empty slot in the path from that leaf to the tree root. The position map which keeps track of the current leaf position for each entry is typically stored on the server side in its own ORAM. This leads to many round trips of interaction for each read/update which is a non-starter for a real-time application such as email.

We note that for email, it is feasible to store the position map client side. As shown in the experiments in Section 4.3, this storage will not exceed 70MB in 10 years for the 95$^{\text{th}}$ percentile user and for most users is closer to 35MB.

Even with the position map stored on the client side, a read or write entails reading everything on the path from a leaf to the root then mapping the read/written entry onto a random leaf and storing it locally in the stash. Finally, the entries read from the path and anything in the stash which is mapped to somewhere on that path is written back. In other words, in Path ORAM (and ORAM in general), entries are shuffled both in case of reads and writes.

At first glance, in case of a lookup in the oblivious index, we can simply omit the complicated machinery for a read of the ORAM (which we only need for reads and writes for index updates) and directly access a single entry in the tree. We do not care if repeated index accesses to the same location are identifiable. However, there are two issues with this. First, the position map only stores what leaf an entry was mapped to, not the particular point along the path from leaf to root where it is stored. This can be fixed by storing, for each keyword, additional information to locate the entry.

The larger issue is that the reshuffling that occurs on a read provides privacy protections not just for the read (which is not important for us) but for subsequent reads and writes. If reads are not shuffled, then an observer can identify when frequently looked up index entries are updated. As a result, we cannot simply have a "half-ORAM": to get completely oblivious writes, we must at some point reshuffle the locations we read from.

Crucially, in our obliviously updatable index, we need not do this on every read (as in ORAM), rather we can defer the shuffling induced by a non-oblivious read to the beginning of an update. We call these **deferred reads**.

This enables considerable savings. First, since updates can be batched (i.e., we collect a bunch of updates to various entries locally and only commit them to the server later), we can shift the computational and bandwidth load to idle time (e.g. when a mobile device is plugged in and connected to wifi) at the cost of temporary client storage. Second, repeated searches for the same term only result in one deferred read. Third, searches for terms that are mapped to the same leaf also only result in one shuffle operation. Finally, because paths overlap even for distinct leaves, we will realize considerable savings: e.g. for 10 deferred read shuffles, we will end up transmitting the root of the tree once instead of 10 times, the children of root twice instead of 5 times, etc. Looking forward to our evaluation, this results in over 90% savings in accesses compared to the simple Path

ORAM.

We note that write-only ORAM constructions [2] do not solve our problem. Write-only ORAM is used in settings where reads leave no record (e.g. where an adversary only has access to snapshots of an encrypted disk, which reveals changes due to writes but not reads.). In these settings, the initial read needed to determine the contents of the block being appended to can be done in the clear. We cannot do that here since we must request the partial bock from the server before appending to it.

To summarize, our obliviously updatable index is a modified Path ORAM scheme with the following changes:

- We keep the position map on the client slide and augment the position map to allow us to locate index entries *inside* a given path from leaf to the root.
- On non-oblivious reads: we lookup the entry directly from its position in the tree (i.e. one disk access), but add the leaf to the list of *deferred reads.*
- On batched reads and updates: we read all paths in the set of deferred reads since the last batch updated and all the paths associated with the updates themselves. We then remap and edit the entries on these paths as in standard Path ORAM and write them back to the ORAM at once.

Security Intuition Deferred shuffling for reads ensures that when a non-deferred read/write happens, the system is in the exact same state as it would be in full Path ORAM. This means that after the deferred read, the position map entries are statistically independent of each other, and we retain the security condition for Path ORAM that $\prod_{j=1}^{M} = Pr(position(a_j)) = (\frac{1}{2^L}^{M})$ *for non-deferred operations.* Intuitively, this models the effect of shuffling a deck of cards: no matter what the previous state was and how the deck was rigged, the shuffle has the same effect. Of course, we have leaked substantial information about the prior state of the index, but that leakage is allowed in SSE! Rather than proving this separately, we will capture it in the proof of security for the SSE scheme itself.

3.2 The Full Protocol

Next, we describe our full DSSE scheme for email which is a combination of the OUI described above and a separate index for full blocks. A detailed description follows.

Let $H = (\mathsf{hsetup}, \mathsf{hlookup}, \mathsf{hwrite})$ be a hash table implementation, $E = (KG, Enc, Dec)$ be a CPA-secure encryption scheme and $F : K \times M \rightarrow C$ be a pseudorandom function. Let W be the universe of all keywords, and $L = \log(|W|)$.

Setup. (Figure 3) For simplicity, we assume that the DB is initially empty, and documents are dynamically added. If not, one can run the SSEAdd protocol we describe shortly multiple times to populate the client and server storages with the documents in DB.

The client generates three random keys k_f, k_e, and k_a, one for the PRF F, and the other two for the CPA-secure encryption scheme.

The client and server initialize the partial-block index, i.e., a non-recursive Path ORAM for a memory of size $|W|$. We denote the tree stored at the server by T, and the corresponding stash stored at the client by S. For all references to Path ORAM we use the notation introduced in Section 2.3. The server also sets up an initially empty full-block index, i.e., an append-only hash table that will be used to store full blocks of document IDs.

For every $w \in W$, the client stores in a local hash table the key-value pair $(w, [pos_w, \ell_w, count_w, r_w, B_w])$, where B_w is a block storing IDs of documents containing w (initially empty), pos_w stores the leaf

```
$\langle \sigma, \mathsf{EDB} \rangle \leftrightarrow$ SSESetup$\langle (1^\lambda, \perp), \perp \rangle$:
Client runs $(h_c, \mathsf{M}_c) \leftarrow \mathsf{hsetup}()$ to setup a local hash table.
Server runs $(h_s, \mathsf{M}_s) \leftarrow \mathsf{hsetup}()$ to setup an append-only hash table.
for $w \in |W|$ do
    $pos_w \overset{R}{\leftarrow} \{0, \dots, 2^L\}$
    $count_w, r_w, \ell_w \leftarrow 0,\ B_w \leftarrow \emptyset$
    Client runs $M_c \leftarrow \mathsf{hwrite}(w, [pos_w, \ell_w, count_w, r_w, B_w], \mathsf{M}_c)$
end for
$k_f \leftarrow K(1^\lambda),\ k_e \leftarrow KG(1^\lambda),\ k_a \leftarrow KG(1^\lambda)$
Client and server run the setup for a non-recursive Path ORAM. Server stores the tree $T$, and client stores the stash $S$.
Client outputs $\sigma = (\mathsf{M}_c, S, k_f, k_a, k_e)$
Server outputs $\mathsf{EDB} = (M_s, T)$
```

Figure 3: Setup for our DSSE scheme

position in $\{0, \dots, 2^L\}$ corresponding to w (chosen uniformly at random), ℓ_w stores the level of the node on path $P(pos_w)$ that would store the block for documents containing w (initially empty), $count_w$ stores the number of full blocks for keyword w already stored in the append-only hash table (initially 0), and r_w is a bit indicating whether keyword w is searched since the last push of the client's block to Path ORAM (initially 0).

The client's state σ will be the hash table M_c, the stash S for the Path ORAM, and the keys k_e, k_f, k_a.

Search. (Figure 4) The client will store the matching documents in the initially empty set $R = \emptyset$. To search locally, the client first looks up w in its local hash table to obtain $[pos_w, \ell_w, count_w, r_w, B_w]$, and lets $R = R \cup B_w$.

It then asks the server for the bucket in the tree T at node level ℓ_w and on path $P(pos_w)$, i.e., $P(pos_w, \ell_w)$. It decrypts the blocks in the bucket using k_e. If it finds a tuple (w, O_w) in the bucket, it lets $R = R \cup O_w$. If r_w is not yet set, the client lets $r_w = 1$ to indicate that w was searched for.

For $i = 1, \dots, count_w$, the client sends $F_{k_f}(w||i)$ to the server, who looks it up in the append-only hash table and returns the encrypted full block A_w^i. The client decrypts using k_a and lets $R = R \cup A_w^i$. The client then outputs R. See Figure 4 for details.

Update. (Figure 5) Let id_d be the document identifier associated with d. For every keyword w in d, the client looks up w in its local hash and adds id_d to B_w. It then checks whether its local storage has reached the maximum limit max_c or not. If not, the update is done. Else, we need to push all the document blocks to the server.

But before doing so, we need to finish the ORAM access for all reads done since the last push. In particular, for all non-zero r_w's, the client needs to read the whole path $P(pos_w)$, re-encrypt all the buckets using fresh randomness, update pos_w to a fresh random leaf, and write the buckets back to the tree using the Path ORAM approach.

Then, for every non-empty block B_w in its local hash, the client performs a full ORAM write to add the documents in B_w to the ORAM block O_w for the same keyword. If O_w becomes full as a result, max_b documents IDs in the block are removed and inserted into $A_w^{count_w+1}$, and inserted to the append-only hash table using a keyword $F_{k_f}(w||count_w + 1)$. See Figure 5 for details.

```
SSESearch$\langle(\sigma, w), \mathsf{EDB} = (T, M_s)\rangle$:

$R \leftarrow \emptyset$
$[pos_w, \ell_w, count_w, r_w, B_w] \leftarrow \mathsf{hlookup}(w, \mathsf{M}_c)$
$R \leftarrow R \cup B_w$
$U \leftarrow$ ReadBucket$(P(pos_w, \ell_w)$
Read $(w, O_w)$ from $U$
$R \leftarrow R \cup O_w$
$r_w \leftarrow 1$
$\mathsf{hwrite}(w, [pos_w, \ell_w, count_w, r_w, B_w], \mathsf{M}_c)$
for $i \in \{1, \ldots, count_w\}$ do
    Client sends $F_{k_f}(w||i)$ to server
    Server returns $C_w^i \leftarrow \mathsf{hlookup}(F_{k_f}(w||i), \mathsf{M}_s)$
    $A_w^i \leftarrow Dec_{k_a}(C_w^i)$
    $R \leftarrow R \cup A_w^i$
end for
Client outputs $R$
```

Figure 4: Search for our DSSE scheme

3.3 Security Analysis

As noted in Section 2, security of an SSE scheme is defined with respect to a leakage function L on the database of documents DB as well as the history of search/update operations in the index. We first specify the leakage function for our construction.

The Leakage Function Recall that $\mathsf{DB} = (d_i, W_i)_{i=1}^N$ is the database of document-keyword pairs and $W = \cup_{i=1}^N W_i$ is the universe of keywords.

During the setup, $\mathsf{L}(\mathsf{DB})$ outputs the size of database $|\mathsf{DB}|$, i.e., the total number of initial document-keyword pairs in the database. For simplicity we can assume this is zero initially. On each search query w_i, the leakage function $\mathsf{L}(\mathsf{DB}_{i-1}, H)$, leaks the pattern of searches for w_i in the history H. Note that this does not leak w_i itself, but only the location of all search/update queries for w_i in the sequence all previous read/updates. On an update query d_i, the leakage function $\mathsf{L}(\mathsf{DB}_{i-1}, H)$ leaks the total number of keywords in d_i, and also the total number of keywords in d_i for which the number of documents in the index has reached the multiple of our designated block-size. Again, this does not leak what the actual keywords are and how they are related to previous searches/updates (no leakage of patterns). This leakage on updates simply captures the fact that the server learns when full-blocks are pushed from the partial-block index to the full-block index.

Theorem 1 *Our SSE Scheme is* L*-secure (see definition in Section 2.4), if* F *is a pseudorandom function, and* E *is a CPA-secure encryption scheme.*

Proof Sketch: First, we need to describe a simulator S that given access to the leakage function describe above, simulates the adversary A (i.e. untrusted server's) view in the real world.

Description of the Simulator: The simulator initializes a local position map that it uses for bookkeeping, just as the honest client would.

On each update query d_i (or many updates if they are batched), the simulator learns the number of keywords in d_i, n. It also learns the total number of keywords in d_i that just reached a full block

```
$\text{SSEADD}\langle(\sigma, id_d), \mathsf{EDB}\rangle$:

for $w \in d$ do
    $[pos_w, \ell_w, count_w, r_w, B_w] \leftarrow \mathsf{hlookup}(w, \mathsf{M}_c)$
    $B_w \leftarrow B_w \cup \{id_d\}$
    $\mathsf{hwrite}(w, [pos_w, \ell_w, count_w, r_w, B_w], \mathsf{M}_c)$
    $size_c \leftarrow size_c + 1$
end for
if $size_c < max_c$ then
    return
else
    $U \leftarrow \{w \in |W| : r_w == 1\}$
    for $w \in U$ do
        $[pos_w, \ell_w, count_w, r_w, B_w] \leftarrow \mathsf{hlookup}(w, \mathsf{M}_c)$
        for $\ell \in \{0, \ldots, L\}$ do
            $S \leftarrow S \cup \text{READBUCKET}(P(pos_w, \ell))$
        end for
    end for
    for $(w, O_w) \in S$ do
        $O'_w \leftarrow O_w \cup B_w$
        if $|O'_w| > max_b$ then
            $count_w \leftarrow count_w + 1$
            $O''_w \leftarrow$ first $max_b$ items in $O'_w$
            $\mathsf{hwrite}(F_{k_f}(w||count_w), O''_w, M_s)$
            $O'_w \leftarrow O''_w - O'_w$
        end if
        $S \leftarrow (S - \{(w, O_w)\}) \cup \{(w, O'_w)\}$
    end for
    for $\ell \in \{L, \ldots, 0\}$ do
        $S' \leftarrow \{(w', O_{w'}) \in S : P(x, \ell) = P(pos_{w'}, \ell)\}$
        $S' \leftarrow$ Select $min(|S'|, Z)$ blocks from $S'$.
        $S \leftarrow S - S'$
        $\text{WRITEBUCKET}(P(x, \ell), S')$
        for $(w, O_w) \in S'$ do
            $\ell_w \leftarrow \ell$
            $r_w \leftarrow 0$
            $B_w \leftarrow \emptyset$
            $pos_w \xleftarrow{R} \{0, \ldots, 2^L\}$
            $\mathsf{hwrite}(w, [pos_w, \ell_w, count_w, r_w, B_w], \mathsf{M}_c)$
        end for
    end for
    $size_c \leftarrow 0$
end if
```

Figure 5: Update for our DSSE scheme

(m) and need to be pushed to the full-block index. It also knows (from the state it is keeping) the location of the leaf for all deferred reads since the last update. The simulator behaves similar to the honest client, except that he generates the leaf location for each keyword being updated uniformly at random and the m locations in the full-block index where the newly filled blocks will go also uniformly at random. It keeps record of all these locations in its position map. Also, for all encrypted blocks it needs to send, it simply generates fresh encryptions of dummy values using a random encryption key it generates.

On each search query for w_i, S learns (from the leakage function) all previous occurrences of search/update for w_i. If this is the first occurrence, it chooses a random entry on the ORAM tree for the OUI, a sequence of random locations that have not yet been looked up in the full-block index, and also stored these locations for bookkeeping. But if it is not the first occurrence, S has previously stored the locations in the partial-block and full-block index it had sent to the server. It simply sends the same locations to the server again. This completes the description of the simulator.

We need to show that the simulated view above is indistinguishable from the view of the adversary in the real execution. We do so using a sequence of hybrids.

$\mathcal{H}_0$: In the first hybrid, the adversary is interacting with the simulator described above. In other words, all "random" locations are generated uniformly at random and independently, and all encrypted values are dummy values.

$\mathcal{H}_1$: In the second hybrid, all locations that were generated uniformly at random are instead generated by the client using a PRF with a random key k_f not known to the adversary.

Note that the advantage of an adversary in distinguishing these two hybrids is bounded by the advantage of an adversary in breaking the PRF security.

$\mathcal{H}_2$: In the third hybrid, all dummy encryptions are replaced by the encryption of actual document identifiers (chosen by the adversary) using a semantically secure symmetric-key encryption as prescribed in the real protocol.

The advantage of the adversary in distinguishing the two games is bounded by the security of the CPA-secure encryption scheme used to encrypt document identifiers.

It only remains to show that the view of the adversary in $\mathcal{H}_2$ is identical to the real protocols described earlier in this Section. We consider adversary's view for the search queries and update queries separately. On update queries, both in the real protocol and in $\mathcal{H}_2$, the location to be updated in the OUI is generated using a PRF as described in the protocol, and so is the location to be added to the full-block index for all keywords that have reached a full block during this update. Similarly, in both cases, the real document keywords used for generating locations and the real document identifiers are encrypted. So the two views are identical.

In case of read queries, if the keyword is being searched for the first time, again the leaf location to be looked up in both the real protocol and H_2 are generated using a PRF, and for repeated reads, in both cases, the exact same locations in the OUI and the full-block index are looked up. Hence the two views are identical. This concludes the proof sketch.

4 Evaluation

In this section we assess the feasibility and performance of our SSE scheme for email, using experiments that are based on real data from a large mail provider. The two important requirements we focus on in our evaluation are (1)**storage usage** both on the server side and the client side, (2)

IO performance on the server side. Again, as existing (non-encrypted) mail search is already IO bound, our primary concern is the second criteria.

4.1 Real World Data

Note that the performance of any SSE scheme critically depends on two main pieces of information: (1) the distribution and frequency of updates to the index, i.e., new keyword-document pairs added to the search index, and (2) distribution and frequency of the keywords being searched.

Data on Index Updates for Email Our email data comes from one of the top three email providers which, for the purpose of anonymized submission, we refer to as nymmail. nymmail has hundreds of millions of monthly active users and maintains a 30 day rolling window of customer email for research. This dataset consists of the sent and received emails of over one hundred thousand users who opted into email collection for research. For privacy reasons, we extract only the frequency of keywords, not the actual words themselves and all analysis of the data itself was performed directly on the map-reduce cluster containing the dataset: only the counts of each token, not the tokens themselves, were exported. On average, users receive 30 emails per day with a standard deviation of 148. Furthermore, each message contains an average of 143 (unstemmed) keywords in it with a standard deviation of 140.

For all keyword data, we strip HTML from the messages and use the standard tokenizer and stopword list from Apache Lucerne.[5] No stemming or other filtering was applied. Between this and the fact that the dataset spanned more than just English, our data represents an upper bound for the "index everything" approach. We identified a total of 62,131,942 keywords in our dataset across all 100K users' email. As the estimated number of words in the English language is in the order of one million, there is ample room for stemming and filtering to drastically reduce this. Nevertheless, we stick with this number for our experiments as an empirical worst case measurement.

Approximating query data For two reasons, we do not have access to comprehensive query data. First, there is no such dataset of users' queries for which users explicitly opted in. Second, existing mail search supports more than single keyword queries and as such queries are often in natural language. It is unclear how to appropriately generate keyword searches from such data. As such, for the sake of experiments, we assume search terms are selected uniformly at random from the set of all indexed words.

Given the low number of queries, we anticipate that the overall effect of this is minimal. The primary concern in our evaluation is the effect of the large number of updates on storage and performance requirements.

4.2 IO Performance of IO-DSSE

We now examine the IO performance of IO-DSSE relative to the operating conditions we observe at nymmail. Our goal is to measure the ability of our scheme to scale out to millions of concurrent instances on arbitrary (and likely proprietary) cloud infrastructure. As a result, we model such an system abstractly as key value store and measure then number of read and writes against it. We implement our scheme in python against against this abstraction.

Our measurements are taken over 30 days of traffic generated using the sampled distribution of keywords discussed above and is repeated for 50 iterations. We assume that the system pushes all client-side email messages to the server every day. This models a mobile device that has access to

[5] We used the list `StopAnalyzer.ENGLISH_STOP_WORDS_SET`.

a free Internet connection when at home. Given that we fix the distribution of keywords, we are left with three variables: the number of searches, the number of emails per day, and the number of keywords per email.

For simplicity, we fix the number of keywords at 350 (this is two standard deviations above the average), and vary the number of email messages starting with the average and incrementing by the standard deviation. This has the net effect of changing the total number of updates per day which (along with the distribution of keywords) is the actual controlling variable for performance. Similarly, we fix the number of searches at one per day, modeling a very active user.

Finally, we somewhat arbitrarily fix the Path ORAM parameters, assuming a height of 17 at 4 buckets per level with each ORAM entry containing a block of 500 identifiers (at 64 bits per file ID, this gives us blocks of 4KB disk blocks). Real deployments should tune these parameters dependent on system architecture and testing. We stress again that our goal is not to see how our system handles large indexes—individual mail inboxes are at most 10s of gigabytes—but it measure the resources used by a small index. This allows us to see how costly it is to deploy in a setting with hundreds of millions of inboxes and thus indexes supported by a minimal number of servers.

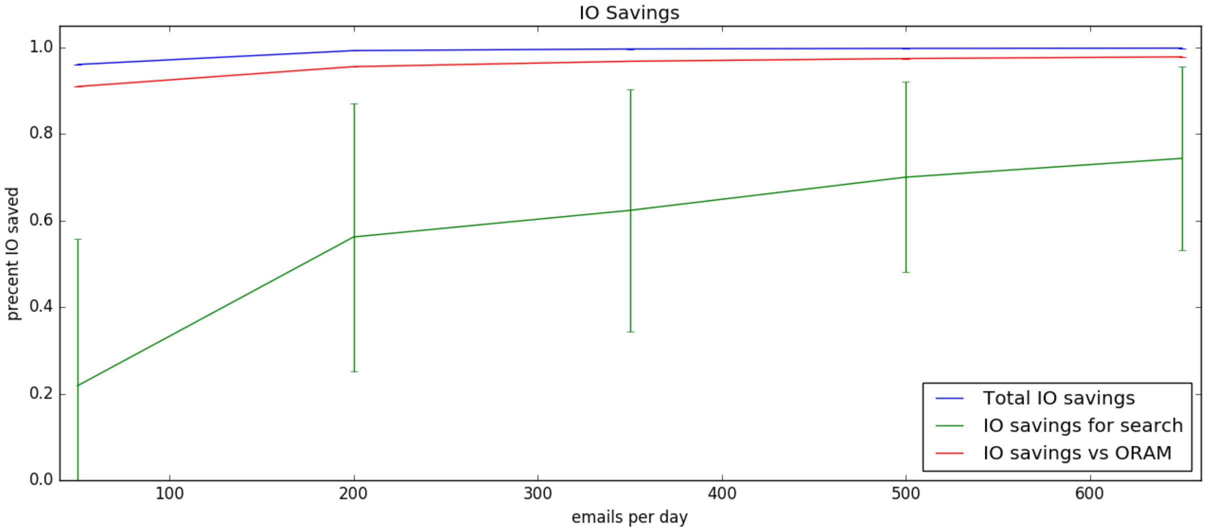

Figure 6: Experimental and simulated results. IO savings for IO-DSSE vs existing approach and vs an obliviously updatable index constructed with PATH ORAM.

Our experiment measures the actual IO savings of our scheme against:

- An encrypted index which, for updates, stores each keyword-document pair at a random location. This represents the state of the art schemes under purely dynamic insertion [6, 20], which perform identically.
- Our solution but with a naive implementation of a Obliviously Updatable Index built with Path ORAM.

Because we are mainly concerned with the IO cost, it was not necessary to implement the comparison systems. For the current state of the art encrypted index, the number of random writes will simply be equal to the number of inserted keywords, and the number of random reads equal to the number of search results. This provides a lower bound on the number of accesses required for all of the DSSE schemes we compare with.

For a solution that uses the Path ORAM scheme as the OUI, the cost of a search is dominated by one ORAM read and one ORAM write. An ORAM "read" would be recorded as many reads by our code as each level in the path generates a distinct access. Thus, instead of one non-deferred read as in our scheme, we charge *oramHeight* reads and *oramHeight* writes. By read or write, we mean the access of a single key-value pair in the underlying server-side data structure.

A third approach we could compare against is a variant of Cash et al. scheme [5] where full-blocks are stored on the server-side, and partial blocks are stored locally on the client side. This would exhibit better performance than our scheme in the short term, because it reduces IO costs compared to an OUI. However, when the client-side storage becomes full, the blocks need to be pushed to the index even if they are only partially full and indeed maybe almost empty. As we will see in the storage discussion, this does not work in practice as the client would need to regularly push partially-filled blocks to the static index, hence defeating the IO efficiency gains of packing many documents identifiers into one block.

Using our implementation, we measure:

Total IO savings: The total amount of IO saved compared to the existing approach of SSE schemes (including [6, 20] under purely dynamic insertion). This includes both search and update.

IO savings for search: The amount of savings on read due to search, ignoring deferred reads. This represents the immediate cost of a search and also the associated latency savings.

IO savings vs. ORAM: Total IO savings when using an OUI vs. ORAM. This is the savings due our optimized obliviously updatable index that does not require full ORAM security.

The results show (see Figure 6) a 99% percent reduction in the IO cost of our scheme compared to a scheme that does one random read per search result and one random write per keyword-document pair. This is slightly different from the naive estimate for the savings of simply batching IO into contiguous blocks of 500 (our chosen block size), i.e., $\frac{x-\frac{x}{500}}{x} = 0.998$ due to both the overhead of access to our obliviously updatable index and the fact that entries will not be packed perfectly, instead being added in smaller groups as they arrive. However, some entries will still be in the stash locally and therefore will not be read from the index. These two issues appear to roughly cancel out.

We show a 94% percent reduction in the reads required by the server for a search query for our scheme vs. simply construction an obliviously updatable index with ORAM. This shows that optimizations stemming from the relaxation of ORAM's security properties are effective.

This is both an important cost saving and a large reduction in the latency of serving the first page results. These results are exactly as predicted, since we do one read instead of 17 (the length of a full path for the chosen parameters) and save $\frac{17-1}{17} = .941$. Finally, our experiments show a 20% to 82% reduction in the IO needed to return a search result. Since search terms are selected uniformly at random from the set of results, many searches have only 1 or 2 documents associated with them, needing only the small corresponding number of reads from the naive index vs. one read from the obliviously updatable index in our scheme. For those, our scheme offers little advantage. As we increase the number of received messages, the total number of indexed documents and therefore the expected number of results per search increases and our scheme becomes more efficient. The variance in the number of results per search term also accounts for the variance in the measured results. We again note that, given the relative infrequence of searches against the index compared to updates, this is not the crucial metric to optimize for.

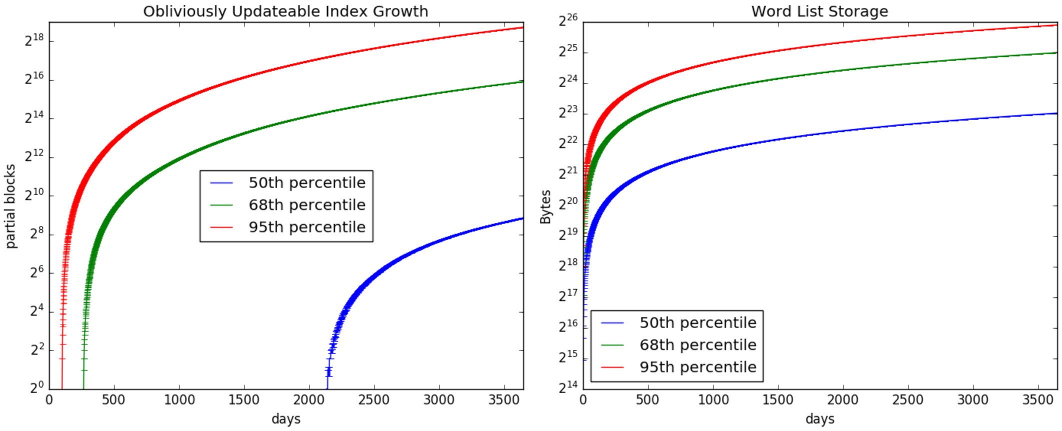

Figure 7: Simulated long term storage for IO-DSSE. Left: the size of the obliviously updatable index needed (i.e. server side storage) as time goes on for users of different activity levels. Right: the size of the client's storage under the same conditions.

4.3 Simulated Long-term Storage Usage

We now examine the storage requirements for IO-DSSE both on the server side (i.e. how large the obliviously updatable index needs to be) and on the client side in terms of stored metadata. The necessary size of the obliviously updatable index is a function of the distribution of keywords, the rate of arrival of keywords, and the amount of available local storage that can be used to buffer results on the client.

Client storage Because the local client is trusted, we are under no constraints as to how the storage is laid out and need not obscure its access pattern. As a result, it does not make a difference if all of the storage is used holding an entry for a single keyword appearing in two million emails or one million keywords each in one email. To a first approximation, client storage is directly proportional to the number of keywords (as we store less than 8 bytes per keyword). As shown in Figure 6, we use $62.7MB \pm 13.2KB$ for a 95$^{\text{th}}$ percentile user and $33.4MB \pm 11.8KB$ for a 50$^{\text{th}}$ percentile user.

Server storage for partial-block index Of course, once the client's storage is full, we must evict entries into the obliviously updatable index, starting with the most full. For OUI, keyword distribution comes into play. At some point the index will be full of infrequent words and we will be forced to evict partial blocks into the full-block index. The main questions we need to answer are 1) how often does this happen and 2) how large of a partial-block index we need to ensure it does not happen too soon.

To measure this directly would involve experiments spanning the expected lifetime of a user's mail account, which is prohibitive to evaluate with a real implementation. Instead, we conduct a Monte Carlo style simulation. We draw words at random, according to the measured distribution of tokens described in the previous section and measure how long it takes before we are forced to evict a partially full block from the OUI. Our simulator merely keeps track of how space is allocated locally (client-side) and in the OUI and at what point each becomes full and forces an eviction. The simulator does not provide actual search results as the intention here is to assess storage requirements.

We assume a local store of 128MB and 64-bit email identifiers. We assume the cost of adding a new keyword to the index is 100 bits. Rather than specifying a fixed size for the OUI, we simply measure how many blocks would be needed to fit the entire index.

Our simulation validates that the general approach of splitting the index into a partial-block and full-block index is viable, showing that even for a 95th percentile user, 2GB of partial-block index is sufficient to hold *ten years* worth of email or nearly 100 years of email for the average user.

As the graph in Figure 6 shows, however, there appears to be no upper bound on the size of the index, short of the total number of observed words. This validates the following intuition: evicting frequent words from the OUI when the block is full ultimately does not free enough space for infrequent words. That space will immediately be occupied by (in many cases the same) frequent word. The long tail of keywords causes problems. To accommodate this, providers must either (1) limit the number of indexed words (e.g. to English words), or (2) limit the amount of indexed email. In the case of a limited set of keywords, the obliviously updatable index would have a fixed size.

Deletes Recall, we can only delete emails from the obliviously updatable index, so any index entries that have been evicted from the OUI are permanent. How long does this give us to fully delete a message (i.e. all index entries resulting from that message's arrival)? For a 95th percentile user, the most frequent word is evicted 6724.8 ± 3.22490 times in 3650 days or 2 evictions per day. For such a user, all index entries for an email can only be deleted within a day of arrival. For the 50th percentile user, on the other hand, where the most frequent word is evicted 319.4 ± 0.843274 times in 3650 days, all index entries for an email can be removed if the email is deleted within an expected 11 days of arrival.

5 Related work and attacks

5.1 Related work on searchable encryption

Searchable encryption has been studied in an extensive line of works [6, 7, 8, 10, 15, 19]. Very few works have focused on efficiency or locality. Cash et al. [6] provide the best such approach. As we stressed in the introduction, however, this approach does not help in the dynamic case: all existing techniques insert each document ID associated with a given keyword into a random location in the index.

5.2 Attacks

Recently Zang et al. [23] construct a highly effective query recovery attack on SSE schemes. The attack leverages the fact that an attacker who can insert entries into the index can construct them such that the subset of attacker files returned uniquely identifies the queried keyword. Effectively, this is a more efficient version of the attacker inserting one unique file per possible keyword which contains only that single keyword. As this attack requires an adversary to insert files into the index, it cannot be mounted by an adversary who wishes to passively surveil many users. None-the-less, our scheme is subject to the attack, and indeed if used for email, highly susceptible due to the ease with which an attacker can insert files. This makes the scheme useful for end-to-end encryption settings which protect against dragnet surveillance. However, it should not be used in scenarios where greater protection again an actively malicious mail server is needed. Also, as the attack depends only on observing retrieved files, there appears to be no simple countermeasures. Forward private SSE schemes [4] do thwart the adaptive version of the attack that requires injecting fewer files, but again at a large locality cost. Moreover as Zang et. al make clear, the non-adaptive attack still works even with forward privacy and that attack is readily mountable in the setting of email.

We are hopeful that some method of injecting noise into the results or merely detecting when this attack has been mounted will be developed. As these techniques likely operate over the logical structure of the index, our scheme should be fully compatible with them. But absent such countermeasures, SSE in the email setting is only secure against passive adversaries regardless of its IO efficiency.

6 Extensions and Conclusions

Encrypted search for email or similar messaging systems represents a major obstacle for E2E encrypted applications. All existing built solutions place a prohibitively high IO cost on updating the index on message arrival: requiring one random write per document-keyword pair and one random read per search. Using a hybrid approach where updates are done to a dynamic ORAM-like index and then evicted to a chunked index typically used for static searchable encryption, we are able to reduce the total IO usage by 99%, and by building a dynamic index that does not protect read privacy, we are able to achieve a 94% reduction in the upfront costs of search.

Our approach, of course, is still more expensive than non-encrypted search, and deploying for email is, in the end, a cost-benefit analysis between the value of protecting user privacy and the operating cost. But this is at least now a trade off that is far easier to make given our performance improvements. Indeed, without such a reduction in IO cost, the cost of encrypted search for email is too high.

Achieving this comes at some cost. First, we must slightly relax the leakage function for searchable encryption: an attacker learns when entries are moved from the partial to full-block index, and we leak slightly more to an active attacker. Second, at present, we only support deletes from the obliviously updatable index, and third, we only provide single keyword search.

Future work and extensions The techniques of Cash et. al [6] can readily be applied to our approach to get conjunctive search. We can simply use their (or any other similar) scheme directly for the full-block index. In their scheme, a keyword is associated with a key used specifically for computing intersection tags $xtag$ based on indexes and set of all such tags $xSET$ is stored by the server. Because no additional data is associated with the index entries on the server, and tags can safely be added to $xSET$ without additional leakage, this technique can be applied to our approach simply by having the server store the tag set.

To deal with deletes, it is possible to incrementally rebuild the index. Instead of appending to the full index on eviction, we can with some small probability overwrite a block in the full index with one evicted from the partial index, storing the overwritten block locally and then incrementally feeding the non-deleted entries back. Thus the entire index would periodically be refreshed. However, careful analysis of the specific rate of deletion is needed to check if this approach provides any practical benefit.

Finally, it is an interesting question weather similar relaxations to ORAM security can be used to build an obliviously updatable index with something other than Path ORAM, e.g. using the techniques of [21] or [1].

References

[1] Bindschaedler, V., Naveed, M., Pan, X., Wang, X., and Huang, Y. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 837–849.

[2] Blass, E.-O., Mayberry, T., Noubir, G., and Onarlioglu, K. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 203–214.

[3] Boneh, D., Mazieres, D., and Popa, R. A. Remote oblivious storage: Making oblivious ram practical.

[4] Bost, R. Sophos - forward secure searchable encryption. Cryptology ePrint Archive, Report 2016/728, 2016. `http://eprint.iacr.org/2016/728`.

[5] Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Rosu, M.-C., and Steiner, M. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS'14)* (2014).

[6] Cash, D., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M.-C., and Steiner, M. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 353–373.

[7] Chase, M., and Kamara, S. Structured encryption and controlled disclosure. In *Advances in Cryptology-ASIACRYPT 2010*. Springer, 2010, pp. 577–594.

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

[9] Damgård, I., Meldgaard, S., and Nielsen, J. B. Perfectly secure oblivious ram without random oracles. In *Theory of Cryptography*. Springer, 2011, pp. 144–163.

[10] Goh, E.-J., et al. Secure indexes. *IACR Cryptology ePrint Archive 2003* (2003), 216.

[11] Goldreich, O. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing* (1987), ACM, pp. 182–194.

[12] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM) 43*, 3 (1996), 431–473.

[13] Kamara, S., and Papamanthou, C. Parallel and dynamic searchable symmetric encryption. In *Financial cryptography and data security*. Springer, 2013, pp. 258–274.

[14] Kamara, S., Papamanthou, C., and Roeder, T. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 965–976.

[15] Naveed, M., Prabhakaran, M., and Gunter, C. A. Dynamic searchable encryption via blind storage. Cryptology ePrint Archive, Report 2014/219, 2014. `http://eprint.iacr.org/`.

[16] Pappas, V., Krell, F., Vo, B., Kolesnikov, V., Malkin, T., Choi, S. G., George, W., Keromytis, A., and Bellovin, S. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 359–374.

[17] Popa, R. A., Redfield, C., Zeldovich, N., and Balakrishnan, H. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 85–100.

[18] Shi, E., Chan, T.-H. H., Stefanov, E., and Li, M. Oblivious ram with o ((logn) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*. Springer, 2011, pp. 197–214.

[19] Song, D. X., Wagner, D., and Perrig, A. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on* (2000), IEEE, pp. 44–55.

[20] Stefanov, E., Papamanthou, C., and Shi, E. Practical dynamic searchable encryption with small leakage. *IACR Cryptology ePrint Archive 2013* (2013), 832.

[21] Stefanov, E., and Shi, E. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on* (May 2013), pp. 253–267.

[22] Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., and Devadas, S. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 299–310.

[23] Zhang, Y., Katz, J., and Papamanthou, C. All your queries are belong to us: The power of file-injection attacks on searchable encryption. Cryptology ePrint Archive, Report 2016/172, 2016. http://eprint.iacr.org/2016/172.