Searching Encrypted Data with Size-Locked Indexes

Min Xu, *University of Chicago;* Armin Namavari, *Cornell University;*
David Cash, *University of Chicago;* Thomas Ristenpart, *Cornell Tech*

https://www.usenix.org/conference/usenixsecurity21/presentation/xu-min

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the
30th USENIX Security Symposium
is sponsored by USENIX.

Searching Encrypted Data with Size-Locked Indexes

Min Xu
University of Chicago

Armin Namavari
Cornell University

David Cash
University of Chicago

Thomas Ristenpart
Cornell Tech

Abstract

We investigate a simple but overlooked folklore approach for searching encrypted documents held at an untrusted service: Just stash an index (with unstructured encryption) at the service and download it for updating and searching. This approach is simple to deploy, enables rich search support beyond unsorted keyword lookup, requires no persistent client state, and (intuitively at least) provides excellent security compared with approaches like dynamic searchable symmetric encryption (DSSE). This work first shows that implementing this construct securely is more subtle than it appears, and that naive implementations with commodity indexes are insecure due to the leakage of the byte-length of the encoded index. We then develop a set of techniques for encoding indexes, called *size-locking*, that eliminates this leakage. Our key idea is to fix the size of indexes to depend only on features that are safe to leak. We further develop techniques for securely partitioning indexes into smaller pieces that are downloaded, trading leakage for large increases in performance in a measured way. We implement our systems and evaluate that they provide search quality matching plaintext systems, support for stateless clients, and resistance to damaging injection attacks.

1 Introduction

Client-side encryption protects data stored at untrusted servers, but deploying it poses both usability and security challenges. Off-the-shelf file encryption disables server-side data processing, including features for efficiently navigating data at the request of the client. And even with well-designed special-purpose encryption, some aspects of the stored data and user behavior will go unprotected.

This work concerns text searching on encrypted data, and targets replicating, under encryption, the features provided in typical plaintext systems efficiently and with the highest security possible. Diverse applications are considered, but a concrete example is a cloud storage service like Dropbox, Google Drive, and iCloud. These systems allow users to log in from anywhere (e.g., from a browser) and quickly search even large folders. The search interface accepts multiple keywords, ranks the results, and provides previews to the user. To provide such features, these storage services retain access to plaintext data. In contrast, no existing *encrypted* storage services (e.g., Mega, SpiderOakOne, or Tresorit) supports keyword search.

The problem of implementing practical text search for encrypted data was first treated by Song, Wagner, and Perrig [40], who described several approaches. Subsequently a primitive known as *dynamic searchable symmetric encryption (DSSE)* was developed over the course of an expansive literature (c.f., [7–9, 11, 13–17, 25–27, 31, 41, 46]). But DSSE doesn't provide features matching typical plaintext search systems, and more fundamentally, all existing approaches are vulnerable to attacks that recover plaintext information from encrypted data. The security of DSSE is measured by leakage profiles which describe what the server will learn. *Leakage abuse attacks* [6, 10, 18, 19, 21, 33, 38, 43, 45, 48] have shown that DSSE schemes can allow a server to learn substantial information about the encrypted data and/or queries. Even more damaging have been *injection attacks* [10, 48], where adversarially-chosen content (documents or parts of documents) are inserted into a target's document corpus. The adversary can identify subsequent queried terms by observing when injected content matches searches, which is revealed by the leaked *results pattern* of DSSE schemes.

Contributions. This work returns to a simple, folklore approach to handling search over encrypted documents: simply encrypting a standard search index, storing it remotely and fetching it to perform searches. In fact this idea was first broached, as far as we are aware, by the original Song, Wagner and Perrig [40] paper, but they offered no details about how it would work and there has been no development of the idea subsequently. While the approach has many potentially attractive features, including better security and the ability to provide search features matching plaintext search, it has not received attention perhaps because it seems technically uninteresting and/or because the required bandwidth was thought impractical — indexes can be very large for big data sets.

We initiate a detailed investigation of encrypted indexes. Our first contribution is to show the insecurity of naively encrypting existing plaintext search indexes, such as those produced by the industry-standard Lucene [1]. The reason is that Lucene and other tools use compression aggressively to make the index — a data structure that allows fast ranking of documents that contain one or more keywords — as compact as possible. Compression before encryption is well known to be dangerous, and indeed we show how injection attacks would work against this basic construction.

We therefore introduce what we call *size-locked indexes*. These are specialized indexes whose representation as a bit string has length that is a fixed function of information we allow to leak. We show a compact size-locked index whose length depends only on the total number of documents indexed and the total number of postings handled. By coupling our size-locked index with standard authenticated encryption, we are able to build an encrypted index system that works with stateless clients and provides better search functionality (full BM25-ranked search) than prior approaches, while resisting both leakage abuse and injection attacks. We provide a formal security model and analysis.

Our encrypted size-locked index already provides a practical solution for moderately sized document sets. But for larger document sets it can be prohibitive in terms of download bandwidth, for example providing a 228.15 MB index for the full 1.7 GB classic Enron email corpus. Here prior techniques like DSSE require less bandwidth to perform searches. We therefore explore optimizations to understand whether encrypted indexes can be made competitive with, or even outperform, existing approaches.

We show two ways of partitioning our size-locked indexes to reduce bandwidth. Our vertical partitioning technique exploits the observation that, in practice, clients only need to show users a page of results at a time. We therefore work out how to securely partition the index so that the top ranked results are contained within a single (smaller) encrypted index, the next set of results in a second-level index, and so on. Handling updates is quite subtle, because we must carefully handle transferring data from one level to another in order to not leak information in the face of injection attacks. We provide an efficient mechanism for updates. We show formally that vertical partitioning prevents injection attacks and only leaks (beyond our full index construction) how many levels a user requested. Because most users are expected to most often need only the first level, vertical partitioning decreases average search bandwidth by an order of magnitude.

We also consider horizontal partitioning which separates the space of keywords into a tunable parameter P of partitions, and uses a separate vertically partitioned size-locked index for each. This gives us a finely tunable security/performance trade-off, since now performing searches and updates can be associated by an adversarial server to certain partitions. We also give an approach to progressively partitioning an index so that the leakage can be gradually increased to maintain performance. We formally analyze the security achieved, and heuristically argue that for small P our scheme's leakage is less dangerous than the result patterns revealed by prior approaches. In terms of performance, horizontal plus vertical partitioning enable us to match the bandwidth overheads of DSSE. For example with the full Enron corpus indexed, our construction using vertical partitioning combined with just 10 horizontal partitions is able to fetch the first page of results for a search in 690 ms and using 7.5 MB of bandwidth.

2 Problem Setting and Background

We target efficient, secure encrypted search for cloud services such as Dropbox and Google Drive.

Search features. We briefly surveyed search features of several widely used storage services. Some features are not precisely documented, in which cases we experimentally assessed functionality. Some details appear in Appendix A.

Evidently, search features vary across plaintext services. Common features include conjunctive (but not disjunctive) queries, relevance ranking of some small number k of returned results, and updating search indices when keywords are added to documents. Interestingly, none of these services appear to update indices when a word is removed from a document.

All surveyed services supported both an application that mirrored data as well as *lightweight* (web/mobile) portals. When the data is locally held by the client, search is easy (say, via the client system's OS tools). For lightweight clients, search queries are performed via a web interface with processing at the server. Previews of the top matching documents are returned, but documents are not accessed until a user clicks on a returned link for the matching document. A user can also request the subsequent pages of results.

In summary, our design requirements include:

- Lightweight clients with no persistent state should be supported, so users can log in and search from anywhere.
- Multi-keyword, ranked queries should be supported.
- Search results may be presented in pages of k results ($k \approx 10$) with metadata previews including name, date, and size.
- The addition and deletion of entire documents should be supported. Deletion of words from documents is optional.

For simplicity, we will assume a single client, but note that our techniques can support multiple clients in the full version. Many plaintext indexes do not decrease in size upon word and document deletion. Looking ahead, our indexes will similarly not decrease in size due to deletions or modifications.

Threat model. Encrypted cloud storage should ensure the confidentiality of a user's data and queries, even when the service is compromised or otherwise malicious.

Leakage-abuse and injection attacks work against existing approaches to practical encrypted search, such as DSSE.

In leakage-abuse attacks, first explored by Islam et al. [21] and Cash et al. [10], the adversary obtains access to all (encrypted) data stored at the server, as well as a transcript of (encrypted) search queries. All DSSE schemes have some leakage, such as the results pattern mentioned above. Given also some side information about the distribution of keywords across documents, prior work has shown that the results pattern leakage is often sufficient to identify queries and, in turn, partial information about document plaintext. A long line of subsequent work has explored various forms of leakage-abuse attacks [6, 18, 19, 33, 38, 43, 45, 48].

Leakage-abuse attacks are typically passive, in the sense that the adversary observes queries but does not actively manipulate documents or queries. Injection attacks instead have the adversary combine observations of encrypted storage and queries with the ability to force the client to insert documents and/or make queries. They were first briefly suggested by Cash et al. [10] and later explored in depth by Zhang et al. [48]. When combined with results pattern leakage, an attacker who can inject chosen documents with known keywords will know when a subsequent (unknown) search matches against the injected document.

We discuss related work in detail in Section 7. The current state of affairs is that we do not currently have systems for encrypted searching that (1) come close to matching the functionality of contemporary plaintext search services; (2) that work in the required deployment settings, including lightweight clients; and (3) that resist these classes of attacks.

Information retrieval definitions. Our work will build off standard information retrieval (IR) techniques. Here we recall some necessary details, and refer the reader to [30, 49] for more extensive overviews.

A *term* (equivalently, keyword) is an arbitrary byte string. A *document* is a multiset of terms; this is commonly called the "bag of words" model. This formalism ignores the actual contents of the document and only depends on the terms that are output by a document parser and stemmer. We assume all documents have an associated unique *identifier* that is used by the underlying storage system, as well as a small amount of *metadata* for user-facing information (e.g. filename or preview). Looking ahead we will assume 4-byte identifiers.

The *term frequency* of a term w in document d, denoted $\mathsf{tf}(w,d)$, is the number of times that w appears in d. In a set of documents $D = \{d_1, d_2, \ldots\}$, the *document frequency* of a term w, denoted $\mathsf{df}(w,D)$, is the number of documents in D that contain w at least once.

A *query* is a set of terms. The most popular approach to ranking search results assigns a positive real-valued score to every query/document pair, and orders the documents based on the scores. We use the industry standard ranking function BM25 [39]. For a query q, set of documents D, and document $d \in D$, the BM25 score is

$$\mathsf{BM25}(q,d,D) = \sum_{w \in q} \log\left(\frac{|D|}{\mathsf{df}(w,D)+1}\right) \frac{\mathsf{tf}(w,d)\cdot(k_1+1)}{\mathsf{tf}(w,d)+k_1\left(1-b+b\frac{|d|}{|d|_{avg}}\right)},$$

where $|d|$ ($|d|_{avg}$) is the (average) document length (where length is simply the size of the multiset); k_1 and b are two tunable parameters, usually chosen in $[1.2, 2.0]$ and as 0.75, respectively. We note that to compute the BM25 score of a query for a given document, it is sufficient to recover the document frequencies of each term in the search along with the term frequencies in the document for each term.

The standard approach for implementing ranked search is to maintain an *inverted index*. These consist of per-term precomputed *posting lists*. The posting list contains some header information (e.g., the document frequency), and then a list of *postings*, each of which *records the occurrence of the term in a document*, usually including the document identifier along with extra information for ranking (for BM25, the term frequency). In our notation, a posting list for term w is written as follows:

$$\mathsf{df}(w,D)\,,\ (\mathsf{id}_1, \mathsf{tf}(w,d_1)), \ldots, (\mathsf{id}_n, \mathsf{tf}(w,d_n))$$

where n is the number of documents that w appears within. A search is processed by retrieving the relevant posting lists (multiple ones in case of multi-keyword queries), computing the BM25 scores, and sorting the results. To improve latency, posting lists are usually stored in descending order of term frequency or document identifier. The latter improves multi-term search efficiency, while the former allows for easily retrieving the most relevant results for a single term search.

In practice, inverted indexes are highly amenable to and compression (c.f. [30], Chapter 5). Many mature search tools are available. For example Lucene [1] is a popular high-performance text search engine that we will use below.

3 Insecurity of Encrypting Standard Indexes

We take a closer look at an idea briefly mentioned by Song, Wagner and Perrig [40]: just encrypt a standard index and store it at the server. We flesh out some details and then show that using this approach with standard tools can be vulnerable to document injection attacks (and possibly more). The key observation is that changes in the length of the encrypted index blob will depend on the number of keywords present in the index in an exploitable way.

Naive encrypted indexing. A simple approach to adding search to an outsourced file encryption system, such as those discussed in Section 2, is to have a client build a Lucene (or other standard) index, encrypt it using their secret key, and store it with the service. To search, the client downloads the index, decrypts, and performs searches with the result. Should client state be dropped (e.g., due to closing a browser or flushing its storage), the next search will require fetching the encrypted index again.

To update the index in response to new files being added or changed, the client can download the encrypted index (if not already downloaded), update it, re-encrypt, and upload. This approach may not scale well with large indices, but at least it

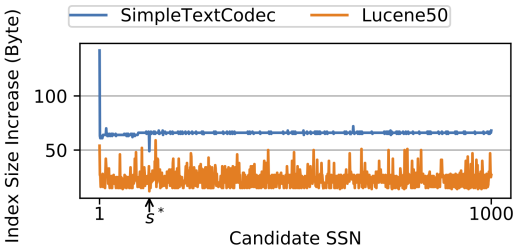

Figure 1: Document-injection attack on Lucene to recover indexed term. The term s^* results in a noticeably smaller change in byte-length than other terms.

might seem to be secure: strong randomized encryption hides everything about the index except its length as an encoded plaintext byte string.

Case study: Lucene. To explore whether leaking plaintext byte-lengths can be exploited, we built a simple encrypted index using Lucene. We give more details about Lucene and our configuration of it in Appendix B. We use two different Lucene encodings, the naive `SimpleTextCodec` and `Lucene50` (the default). The adversary is given the byte-length of the plaintext encoding after updates. For `SimpleTextCodec` only one file is output, and we use its size. With `Lucene50` several files are output, so we use the total sum of their sizes. This captures the assumption that encryption leaks the exact length of the plaintext data, which would be true if one uses any standard symmetric encryption scheme, e.g., AES-GCM and ChaCha20/Poly1305.

We considered the following file-injection attack setting: An index has been created that contains a single document containing exactly one term which is a random 9-digit numerical, e.g., a social security number (SSN) that we denote s^*. An adversary is given a list of 1,000 random SSNs $s_1, \ldots, s_{1000}$, one of which equals s^*, and the adversary's goal is to determine which s_i equals s^*. Our attacker is allowed to repeatedly inject a document of its choosing and observe the new byte-length of the index. A secure system should keep s^* secret, even against this type of adversary.

Our attacker works as follows: it records the initial byte-length of the index. Then for each of the 1,000 SSNs in its list, the attacker injects a document consisting of exactly that SSN. It then records the change in byte-length of the index. (Documents are not deleted here, so at the conclusion of this attack, the index contains 1,001 documents.) Finally, the attacker finds the injected SSN that resulted in the *smallest* change in byte-length, and uses that as its guess for s^*. The intuition is that adding a new keyword to the index increases its size more than adding a new posting for an existing keyword.

We plot two example runs in Figure 1, one for each encoding. The horizontal axis corresponds to the injected terms in order, and the vertical axis is the change in byte-length after each injection. We observe first that our attack worked for both encodings, since the smallest change corresponded to s^* (as is visible in the plot). We also observe that this worked despite quite a bit of noise, especially in case of `Lucene50`, where the variation in changes due to the internals of Lucene is visible. We repeat the attack 100 times, each with a different s^* from the 1000 candidate SSNs, and the attack succeeded every single time.

Discussion. While our example above is very simple, we expect that other attacks exploiting length leakage are possible. For instance, an index may compress its term dictionary, so injecting terms similar to existing terms may result in a different byte-length change than injection terms that are far from the existing dictionary. We have also not exploited the variable-byte encoding used in postings lists.

On the other hand, a tempting seeming fix would be to combine length-hiding authenticated encryption (LHAE) [36] schemes with padding of the full index to prevent plaintext length leakage. But it's unclear a priori how to efficiently pad in a way that would prevent attacks.

We conclude that a well-controlled approach to sizing indexes is required to have confidence in the security of this kind of encrypted index approach.

4 Secure Encrypted Indexes

As shown in the last section, subtle issues can expose encrypted indexes to attacks. Intuitively, we need to ensure that server-visible information, such as ciphertext lengths and access patterns reveal nothing damaging. For example, preventing file injection style attacks requires ensuring that the precise number of keywords at any point in time is not revealed.

To make this rigorous and guide our designs, we formalize the notion of an encrypted index. Our approach gives formal syntax, semantics, and security for search mechanisms that reflect application requirements in practice (see Section 2).

Encrypted index formalism. Architecturally, an encrypted index involves a client and server, the latter being the storage service. Formally, we abstract the role of the server as an oracle Srv with two interfaces. The Srv.put interface takes as input a pair of bitstrings $(\texttt{lbl}, v)$ and stores as a table entry $\texttt{T}[\texttt{lbl}] \leftarrow v$. One can delete a value by $\mathsf{Srv.put}(\texttt{lbl}, \perp)$. The Srv.get interface takes as input a bitstring `lbl` (the label) and returns $\texttt{T}[\texttt{lbl}]$ (which could be empty, i.e., $\texttt{T}[\texttt{lbl}] = \perp$). We overload notation and allow $(v_1, v_2) \leftarrow \mathsf{Srv.get}(\texttt{lbl}_1, \texttt{lbl}_2)$ to denote fetching the values stored under labels $\texttt{lbl}_1, \texttt{lbl}_2$. The Srv.app interface allows appending a value to a table entry, i.e, $\mathsf{Srv.app}(\texttt{lbl}, v)$ sets $\texttt{T}[\texttt{lbl}] \leftarrow \texttt{T}[\texttt{lbl}] \,\|\, v$ where $\|$ denotes some unambiguous encoding of the concatenation of the existing values and v.

We can now formalize the client side functionality. An *encrypted index scheme* consists of two algorithms Search and Update. Associated to any scheme are four sets: the key space KeySp, identifier space IDSp, metadata space MetaSp, and relevance score space RelSp. Identifiers allow linking a search to a document, metadata includes relevant information about a document that a search should return to help a user

(e.g., document name), and rankings are numerical values indicating a document's relevance score. A secret key, typically denoted K, is chosen uniformly from KeySp. The (randomized) algorithms (which both have access to the server oracle Srv). fit the following syntax:

- $\mathsf{Search}_K^{\mathsf{Srv}}(q,\mathsf{st}) \to (G,\mathsf{st})$ takes as input a key $K \in \mathsf{KeySp}$, a query q (which is a set of bitstrings), and a client-side state value st that is possibly empty ($\mathsf{st} = \varepsilon$). Search outputs a result set G consisting of triples from $\mathsf{IDSp} \times \mathsf{MetaSp} \times \mathsf{RelSp}$ along with new state st.
- $\mathsf{Update}_K^{\mathsf{Srv}}(\Delta,\mathsf{st}) \to \mathsf{st}$ takes as input a $K \in \mathsf{KeySp}$, an *update* Δ and a state. We define updates to be a tuple of the form $(\mathsf{id},\mathsf{md},V)$, where $\mathsf{id} \in \mathsf{IDSp}, \mathsf{md} \in \mathsf{MetaSp}$ and V is a set of term/count pairs, i.e. members of $\{0,1\}^* \times \mathbb{N}$. The algorithm outputs a new state.

The semantics of Search oracle are as follows. A query is a set of bit strings representing the keywords; we focus on supporting BM25 queries (see Section 2). See the full version for a discussion of other search types, including fuzzy search. The client can cache state st between searches. We refer to a sequence of operations that evolve the same state as a *session*. A call to Search with $\mathsf{st} = \varepsilon$ is referred to as a *cold-start search*, and initiates a new session. Depending on the construction, a cold-start search may return a complete or partial list of results. Subsequent calls to Search with same evolving state may be used to obtain more results. For example, our construction will respond to a series of searches for the same query with a growing "effort level" that is tracked in the state. Search sessions may include distinct queries q, in which case the client's performance benefits from local caching.

Updates handle adding new documents to the index, removing old documents, or modifying existing ones. Updates also allow client side state, which will be useful for some performance optimizations but our constructions will always push information to the server immediately to reflect the update. This is important should a client drop state; updates are still available at the server.

We will not formally define correctness, but our constructions will ensure that searches reflect the latest updates. As we provide ranked, paginated results, we will measure empirically search quality relative to standard ranking approaches like BM25.

Security. We use a security definition parameterized by a *leakage profile* specifying what is leaked. The adversary $\mathcal{A}$ controls all inputs (queries and updates) excepting the secret key K, and can observe all interactions between the algorithms and the server Srv.

We formalize security via two games. Pseudocode descriptions appear in Figure 2. The first game, $\textsc{Real}_\Pi$, gives an adversary $\mathcal{A}$ oracles Up, Srch, ClrSt for which it can adaptively choose client inputs. We abuse notation and write $((G,\mathsf{st}),\tau) \leftarrow\!\!\$\ \mathsf{Search}_K^{\mathsf{Srv}}(q,\mathsf{st})$ to denote running Search on a state st and letting τ be the transcript of requests and responses

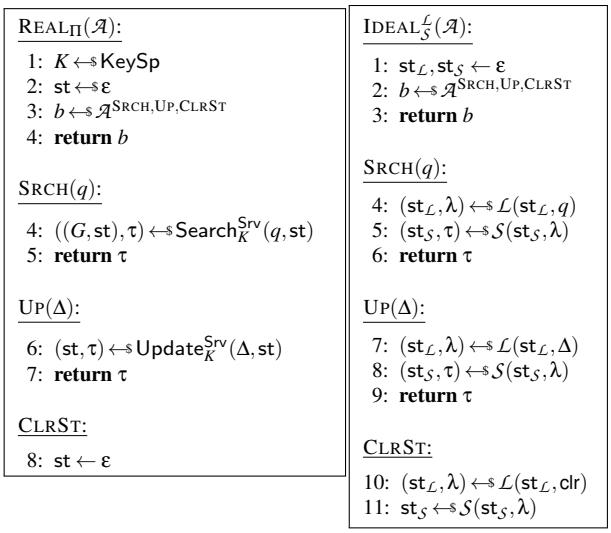

Figure 2: Security games for encrypted index schemes.

made by Search to Srv. We make an analogous overloading of Update. This captures that $\mathcal{A}$ should observe, but not be able to manipulate, all queries to, and responses from, Srv. Note that given the responses τ the adversary can reconstruct the exact state of the server (which is a simple put/get interface that works deterministically).

We allow the adversary to reset the session state st to the empty string by calling the oracle ClrSt, which takes no inputs and produces no outputs. This represents the ending of a session (e.g. logging out of a browser), and subsequent calls will be run with an empty state. The adversary eventually outputs a bit which becomes the output of the game.

The ideal game $\textsc{Ideal}_\mathcal{S}^\mathcal{L}$ is parameterized by a leakage profile $\mathcal{L} = (\mathcal{L}_{se}, \mathcal{L}_{up})$ and a simulator $\mathcal{S} = (\mathcal{S}_{se}, \mathcal{S}_{up})$. In this game the two oracles are instead implemented by a combination of running the appropriate leakage algorithm and handing the resulting input to the respective simulator algorithm. Note that both the leakage and simulator algorithms can be randomized. The simulator algorithms share state; this is made explicit with an input and output bit string $\mathsf{st}_\mathcal{L}$ shared by the algorithms. Ultimately again the adversary outputs a bit, and we let "$\textsc{Ideal}_\mathcal{S}^\mathcal{L}(\mathcal{A}) \Rightarrow 1$" be the event that the output is one, defined over the coins used by the game including those used by $\mathcal{A}$ and $\mathcal{S}$.

Definition 1. *Let $\Pi = (\mathsf{Update},\mathsf{Search})$ be an encrypted index scheme. Let $\mathcal{A},\mathcal{L},\mathcal{S}$ be algorithms. The $\mathcal{L}$-*advantage of *$\mathcal{A}$ against Π and $\mathcal{S}$ is defined to be*
$\mathrm{Adv}_{\Pi,\mathcal{S}}^\mathcal{L}(\mathcal{A}) = |\Pr[\textsc{Real}_\Pi(\mathcal{A}) \Rightarrow 1] - \Pr[\textsc{Ideal}_\mathcal{S}^\mathcal{L}(\mathcal{A}) \Rightarrow 1]|$.

5 Encrypted Index Constructions

In this section we introduce *size-locked encodings* and show how to build secure encrypted indexes using them. We start with our basic encoding then detail a simple encrypted index scheme which performs searches by fetching the entire

index using this encoding. Finally, we show how to scale this construction via two partitioning strategies (*vertical* and *horizontal*) which exchange some limited leakage to improve performance for large document sets.

5.1 Size-Locking Definitions

Our constructions will make use of algorithms to encode updates and merge them into an accumulating index while controlling size-based leakage, and also an algorithm to execute queries on the encodings. We abstract out our basic encoding approach into three algorithms:

- An algorithm $\mathsf{SLEncodeUp}$ that takes as input an update Δ as defined in the syntax of encrypted indexing schemes. It outputs a bytestring U encoding the update.
- An algorithm $\mathsf{SLMerge}$ that takes as input two bytestrings P, U encoding the current index and an update respectively. It outputs a new P bytestring encoding the updated index.
- An algorithm $\mathsf{RunQuery}$ that takes as input a bytestring P encoding an index and a query q. It returns a result set G (following the syntax of Search in the previous section).

We will analyze constructions built from these algorithms and their later extensions generically. In order for maximum expressiveness in constructions, we do not impose a correctness condition, so even trivial versions that output nothing are permitted, but would result in poor search performance. While instantiations will be somewhat lossy, we will show empirically that they provide good search results.

The next two definitions formalize a requirement called *size-locking* for encoding and merging (the querying algorithm $\mathsf{RunQuery}$ will not have any security requirements). Both are parameterized by respective leakage functions $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}$ and $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}$ which describe what input features encoding lengths should depend on. Intuitively, these definitions capture that the output length of update encodings and merging of a sequence of updates are fixed functions of the outputs of the relevant leakage function.

The first definition, for encoding updates, is relatively simple: For any update Δ, the output length of $\mathsf{SLEncodeUp}(\Delta)$ must depend only on the output of $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}$.

Definition 2 (Size-locked updates)**.** *We say an update encoding algorithm* $\mathsf{SLEncodeUp}$ *is* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}$-size-locked *if for all* $\Delta \in D$ *the byte-length of* $\mathsf{SLEncodeUp}(\Delta)$ *is a fixed function of* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}(\Delta)$.

For merging we define a more subtle condition requiring that, for any sequence of updates, the lengths of all the intermediate index encodings is a fixed function of the leakage $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}$ applied to the sequence of updates.

Definition 3 (Size-locked merging)**.** *We say that algorithm* $\mathsf{SLMerge}$ *is* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}$-size-locked *if the following holds for all sequences* $\Delta_1, \ldots, \Delta_r$ *of updates. Define* P_0 *to be the empty string, and for* $i = 1, \ldots, r$ *let* $P_i = \mathsf{SLMerge}(P_{i-1}, \mathsf{SLEncodeUp}(\Delta_i))$. *Then we require that the byte-length of* P_r *is a fixed function of* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}(\Delta_1, \ldots, \Delta_r)$.

This definition implies that lengths of each P_i are determined by $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}(\Delta_1, \ldots, \Delta_i)$, since the condition must also hold for each prefix sequence $\Delta_1, \ldots, \Delta_i$.

We say that an encoding approach is $(\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}, \mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}})$-size-locked if it satisfies both definitions. This definition is convenient for simplifying proofs, since security analyses will rely only on the size function and can otherwise ignore the complexities of encoding.

5.2 Our Size-Locked Encoding

Our leakage functions. Our construction is aimed at the following leakage functions: We want $\mathsf{SLEncodeUp}$ to be $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}$-size-locked for updates where $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}(\Delta)$ outputs the number of postings in Δ For merging, the leakage function $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}(\Delta_1, \ldots, \Delta_r)$ will output the total number of postings added along with the number of documents added In particular we want to avoid leaking, say, when a new unique keyword is added, so care must be taken to hide when an update contains new versus old keywords.

Our encodings. We will describe the accumulated encodings output by $\mathsf{SLMerge}$ first. Afterwards it is simple to describe how our algorithms $\mathsf{SLEncodeUp}, \mathsf{SLMerge}$ maintain this structure as an invariant. Our approach is to follow the structure of traditional search indexes described in Section 2 and maintain posting lists that are encoded to avoid leakage. A merged index will always be a byte string of the form

$$\langle n \rangle_4 \,\|\, \mathtt{bin} \,\|\, \mathtt{fwd} \,\|\, \mathtt{inv},$$

where $\|$ denotes string concatenation, $\langle n \rangle_4$ is an encoding of the number of documents (we use a four-byte representation in our implementation; here and below, we write $\langle v \rangle_k$ for a k-byte encoding of v.), `bin` is a binary lookup table, `fwd` is an encoded forward index byte string, and `inv` is an encoded inverted index byte string. We use two configurable parameters: W is the number of bytes used to encode identifiers and M is the number of bytes of per-file metadata allowed.

Functionally, `fwd` enables mapping document identifiers to their metadata, `inv` is an encoding of the posting lists, and `bin` is some auxiliary data for performance optimization. The goal is to ensure that they together enable efficient computation of search results and have total byte-length equal to a fixed function of the number n of unique documents and the total number of postings N. A summary of the encoding structure is given in Figure 3. The ① and ② stages in Figure 3 illustrate how the full primary index looks with two documents. To limit leakage, the values of n and N will increase monotonically with each update.

Our index encoding will have size exactly

$$(2W+8)\cdot\min\{N, 90\cdot N^{0.5}\} + (W + W/2 + M)\cdot n + (W+1)\cdot N$$

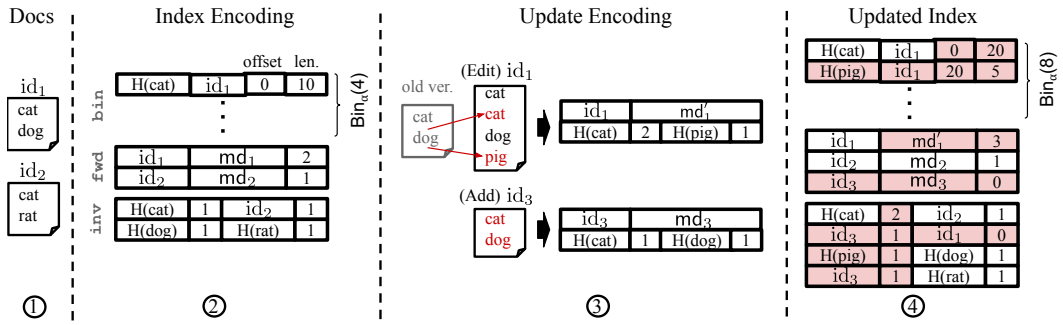

Figure 3: Four stages of size-locked indexing for maintaining the forward (`fwd`) and inverse (`inv`) indexes. ①: documents to be indexed; ②: corresponding `bin`, `fwd` and `inv`; ③: document updates and their encoding; ④: `bin`, `fwd` and `inv` after merging updates with new entries shaded red.

bytes. (We assume that W is a multiple of two, and round the square root in an arbitrary predetermined way.) The first term is the size of `bin`, the second `fwd`, and the second term is the size of `inv`. We will unpack the formula as we explain the components below.

Encoding `inv`. We first describe how `inv` is computed. It will ultimately encode the posting lists independently and concatenate them, so we describe how to encode some posting list $(\mathsf{id}_1, \mathsf{tf}_1), \ldots, (\mathsf{id}_\ell, \mathsf{tf}_\ell)$ for a keyword w, where $\mathsf{tf}_i = \mathsf{tf}(w, \mathsf{id}_i)$ is the term frequency of w in document with id id_i.

We hash terms w to W-byte-long hashes, denoted $H(w)$ (we use a truncated cryptographic hash for H). For small W there can be collisions, in which case we simply merge the colliding posting lists. This happens rarely enough that search accuracy is barely affected in our testing.

Next, we compactly represent term frequencies as one-byte values. This makes BM25 calculations coarser but still enables sufficiently accurate ranking. Specifically we let $\widetilde{\mathsf{tf}}_i$ be a rounding of tf_i to the nearest value expressible in the form $a2^b$, where a, b are four-bit non-negative integers. We thus encode each $\widetilde{\mathsf{tf}}_i$ in one byte, as $a \| b$.

Naively encoding posting lists as a term followed by the list entries would not achieve the size-locking we target, as the output length would depend on the number of terms, enabling injection attacks like those in Section 3. To see this dependence, consider an index with a single term with two postings, versus an index with two terms that have one posting each; Done naively, the latter would have a longer encoding.

We therefore apply a trick to encode a posting list of length ℓ in *exactly* $(W+1)\cdot\ell$ bytes. We enforce domain separation between hashes and document identifiers by fixing the top bit of hashes to be one, i.e., replacing $H(w)$ with $H(w) \vee 10^{8W-1}$ and fixing the top bit of all document identifiers to zero. Then, we remove the first identifier id_1 in each posting list, making it implicit. We will store information in `fwd` to be able to recover it during decoding. Our posting list is encoded as

$$(H(w) \vee 10^{8W-1}) \,\|\, \widetilde{\mathsf{tf}}_1 \,\|\, \mathsf{id}_2 \,\|\, \widetilde{\mathsf{tf}}_2 \,\|\, \cdots \,\|\, \mathsf{id}_\ell \,\|\, \widetilde{\mathsf{tf}}_\ell$$

and `inv` is simply the concatenation of the individual posting lists, in an order to be explained shortly.

Encoding `fwd`. The bytestring `fwd` maps document identifiers to their metadata. Since we allow leaking the number of documents, this could be a relatively simple matter, except that we need to make the first identifiers that were dropped from `inv` recoverable.

The forward index will represent the documents in the order in which they are added, letting $\mathsf{id}_i, \mathsf{md}_i$ be the identifier and metadata of the i^{th} document. Define New_i be the set of terms newly introduced by document id_i and let $ct_i = |\mathsf{New}_i|$, encoded as a $W/2$-byte integer (so ct_i is the number of terms in the i^{th} document but not in any earlier document).

We form `fwd` simply by calculating the string $\mathsf{id}_i \,\|\, \mathsf{md}_i \,\|\, ct_i$ for each document and concatenating them in order. This means $|\texttt{fwd}| = (W + W/2 + M)\cdot n$.

We can now describe the ordering of the post lists in `inv` to enable decoding to recover the omitted first identifiers. We maintain the posting lists within `inv` so that the ct_1 posting lists associated to keywords in New_1 appear first, then the ct_2 lists for New_2, and so on. Thus during decoding we know that the first $ct_1 = |\mathsf{New}_1|$ posting lists should have added id_1, the next $ct_2 = |\mathsf{New}_2|$ should have added id_2, and so on.

Encoding `bin`. It is possible to correctly decode an index from $\langle n \rangle_4 \| \texttt{fwd} \| \texttt{inv}$ only. We however include an extra lookup table `bin` to speed up the recovery of a posting list without fully decoding.

A naive idea is to trade off primary index size (and bandwidth) for search speed by storing a lookup table that maps a word hash to its offset in `inv`. This again fails to meet our size-locking goal, since the serialized index length would depend on the number of terms. Instead we construct the lookup table based on the estimated number of keywords in the document collection, instead of the exact number, following Heaps' law. Heaps' law states that, for a document collection with N postings, the number of unique keywords is roughly $\mathsf{Heaps}_{\alpha,\beta}(N) := \alpha \cdot N^\beta$ (c.f., [30], Chapter 5.1.1). We conservatively choose $\alpha = 90$ and $\beta = 0.5$ to let the estimated number of keywords be an overestimate. We determine the number of postings in `bin` using $\mathsf{Bin}_\alpha(N) = \min\{N, \mathsf{Heaps}_{\alpha,0.5}(N)\}$

Using this, we fix the size of the lookup table to

$$(2W+8)\cdot \mathsf{Bin}_\alpha(N) = (2W+8)\cdot \min\{N, 90\cdot N^{0.5}\}$$

bytes because we store, for each word hash of W bytes, the W-byte identifier of the first document that introduces the word, a four-byte position offset, and a four-byte encoding of the posting list length in bytes.

If the actual number of terms is smaller than the estimate, we add dummy entries to the lookup table. If the number of terms is larger, some words (chosen arbitrarily) will not appear in the lookup table and we can fall back to the linear scan approach.

Encoding algorithm: Updates. Algorithm SLEncodeUp works as follows. An input update Δ consists of a document identifier id, metadata md, and set V of m updated term/term-frequency pairs. (When a keyword is removed from a document, this is represented by a term-frequency of zero.) Here we would like to have an encoding with bytelength determined only by m. To achieve this we simply encode Δ as

$$\mathsf{id} \,\|\, \mathsf{md} \,\|\, H(w_1) \,\|\, \widetilde{\mathsf{tf}}_1 \,\|\, \cdots \,\|\, H(w_m) \,\|\, \widetilde{\mathsf{tf}}_m,$$

where $\widetilde{\mathsf{tf}}_i$ is the single-byte rounding of tf_i as described earlier. For an update with m postings, this encoding will be exactly $W+M+(W+1)m$ bytes. (See stage ③ in Figure 3.)

The following easy claim formalizes the size-locking property of SLEncodeUp.

Claim 4. SLEncodeUp *is* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{up}}$*-size-locked for the function* $\mathcal{L}_{\mathrm{sl}}(\Delta)$ *that outputs* $|V|$*, where* $\Delta = (\mathsf{id}, \mathsf{md}, V)$.

Encoding algorithm: Merging. We finally describe how SLMerge works to maintain the encoding describe above. Recall that SLMerge takes as input an accumulated encoding P and an encoded update U. We can inductively assume that P has the structure described above, namely of the form $\langle n\rangle_4 \,\|\, \texttt{bin} \,\|\, \texttt{fwd} \,\|\, \texttt{inv}$. The encoded update U encodes identifier/metadata/posting-set triple $(\mathsf{id}, \mathsf{md}, V)$ and must be merged into P to produce a new encoding.

We first update `inv`. The set V can contain both keywords that already appear in P and newly introduced keywords. For those that already appear, we simply append the postings to the appropriate lists in `inv` with the encoding described above. For newly-introduced keywords, we create new posting lists with implicit identifiers (following our encoding trick from above), and append them to `inv`.

We next update `fwd` and possibly n. If the updated document identifier id is new, then we increment n and append to `fwd` the string $\mathsf{id}\|\mathsf{md}\|ct$, where ct is the number of new keywords introduced. If id was already in `fwd`, then we update its count ct_i if it introduced any new keywords, and also overwrite its metadata with md.

We finally modify `bin` to reflect for each keyword the new offset and length of its posting list as well as the first document that introduces it, which may have changed as a result of the update. If a document introduces more new words as a result of an addition, we increment ct_i accordingly. If a word that was introduced by the document is removed from the document entirely, we leave ct_i unchanged.

The following claim formalizes the size-locking of SLMerge.

Claim 5. SLMerge *is* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}$*-sized-locked for the function* $\mathcal{L}_{\mathrm{sl}}^{\mathrm{mrg}}(\Delta_1,\ldots,\Delta_r)$ *that outputs* (N,n)*, where if* $\Delta_i = (\mathsf{id}_i, \mathsf{md}_i, V_i)$*, then* $N = \sum_{i=1}^{r} |V_i|$ *is the total number of postings in* $\Delta_1,\ldots,\Delta_r$ *and* $n = |\{\mathsf{id}_1,\ldots,\mathsf{id}_r\}|$ *is the number of unique document identifiers in* $\Delta_1,\ldots,\Delta_r$.

This claim follows by induction on the number of updates. For a single update our encoding obviously only depends on the number of postings. Further updates will increase the length of the encoding by possibly adding a new document entry to `fwd`, and adding a number of bytes to `inv` that depend only the size of V in the update. Finally, the size of `bin` depends only on N.

Querying algorithm. Our implementation of RunQuery simulates the computation of BM25-based ranking. Given a query consisting of possibly several keywords $w_1, w_2, \ldots$, RunQuery uses the offsets in `bin` to find the posting lists and implicit first identifiers for each term. The rounded term frequencies are used in place of real term frequencies, but the rest of the BM25 computation proceeds in the straightforward way.

5.3 Full-Download Construction

Using SLEncodeUp, SLMerge, and RunQuery, we construct the FULL encrypted index scheme in Figure 4. We use standard symmetric encryption with secret key K, with encryption and decryption denoted by Enc_K and Dec_K. We abuse notation in decryption by feeding a vector of ciphertexts as input to mean decrypting each independently. We also abuse notation with our SLMerge algorithm, allowing it to take a vector $\vec{U}$ of encoded updates, which we take to mean running SLMerge repeatedly on each entry along with the accumulated index.

At a high level, the client maintains an index locally that is always either empty or a copy of the full, up-to-date index. The server storage consists of an encrypted, encoded index under some arbitrary unique label `idx` as well as an update cache (under some label `up`) that can be empty. To search, if the client index is empty, the client downloads the entire server state, decrypts, and decodes the results (lines 2–4). Note that we abuse notation slightly, running Dec on a vector, which denotes component-wise application of decryption. After this, the client can run the query and output the results so they can be shown to a user (line 5). Finally, if there were any changes due to updates then the client encodes the updated index, encrypts it, replaces the old encrypted index on the server (recall that Srv.put overwrites old values), and deletes the update cache. (To implement line 6, the client maintains a local dirty bit but we omit this from our notation.)

$\underline{\mathsf{Search}_K^{\mathsf{Srv}}(q,\mathsf{st})}$:

1: **if** $\mathsf{st} = \perp$ **then**
2: $(B,\vec{C}) \leftarrow \mathsf{Srv.get}(\mathtt{idx},\mathtt{up})$ ▷ Fetch encrypted index & updates
3: $P \leftarrow \mathsf{Dec}_K(B)$; $\vec{U} \leftarrow \mathsf{Dec}_K(\vec{C})$ ▷ Decrypt
4: $\mathsf{st} \leftarrow \mathsf{SLMerge}(P,\vec{U})$ ▷ Merge in updates, in any order
5: Output results via $\mathsf{RunQuery}(q,\mathsf{st})$ ▷ Run query
6: **if** $\exists$ outstanding updates **then**
7: $B' \leftarrow \mathsf{Enc}_K(\mathsf{st})$ ▷ Encrypt new index
8: $\mathsf{Srv.put}(\mathtt{idx},B')$; $\mathsf{Srv.put}(\mathtt{up},\perp)$ ▷ Update server

$\underline{\mathsf{Update}_K^{\mathsf{Srv}}(\Delta,\mathsf{st})}$:

10: $U \leftarrow \mathsf{SLEncodeUp}(\Delta)$ ▷ Encode update
11: $C \leftarrow \mathsf{Enc}_K(U)$ ▷ Encrypt update
12: $\mathsf{Srv.app}(\mathtt{up},C)$ ▷ Add to server
13: **if** $\mathsf{st} \neq \perp$ **then**
14: $\mathsf{st} \leftarrow \mathsf{SLMerge}(\mathsf{st},U)$ ▷ Update local state

Figure 4: Full-download encrypted index construction.

To perform an update, the client simply encodes, encrypts, and appends the update information to the update cache (lines 10–12). If the local session state is non-empty, then the client also updates the local state via the routine Merge (line 14), but we do not yet push this changed state to the remote server to save bandwidth. We therefore refer to this type of update as lazy, since we're deferring merging in the update until the next search.

Security analysis. The full-download construction achieves $\mathcal{L}$-adaptive security for a leakage profile $\mathcal{L}$ that reveals only the number of postings in updates and the number of documents added to the system. In particular, updates do not leak if they introduce new terms or not, avoiding the type of attack in Section 3.

In the following theorem, $\mathrm{Adv}^{\text{lor-cpa}}_{\mathsf{SE}}(\mathcal{B})$ refers to the standard definition of advantage for an adversary $\mathcal{B}$ in attacking the left-or-right CPA security of SE.

Theorem 6. *Let* Π *be the encrypted size-locked index scheme in Figure 4 and let* $\mathsf{SE} = (\mathsf{Enc},\mathsf{Dec})$ *be a symmetric encryption scheme. Then there exists a simulator* $\mathcal{S}$ *such that for all* $\mathcal{A}$ *there exists a* $\mathcal{B}$ *such that* $\mathrm{Adv}^{\mathcal{L}}_{\Pi,\mathcal{S}}(\mathcal{A}) \leq \mathrm{Adv}^{\text{lor-cpa}}_{\mathsf{SE}}(\mathcal{B})$*, where* $\mathcal{B}$ *and* $\mathcal{S}$ *run in time that of* $\mathcal{A}$ *plus some small overhead.*

We provide a proof of the theorem in the full version and give a sketch here. The simulator $\mathcal{S}$ will produce transcripts that replace encryptions of index and update encodings with encryptions of all-zero strings that are as long as the encodings. The simulator is able to infer these lengths as a result of size-locking properties and the leakage profile.

Performance optimizations. Our implementation carries out line 4 of searching with some optimizations. To perform a merge, the existing posting list in `inv` is located, and we add all m postings to it. Importantly we do *not* delete prior postings: in the case that there are two postings for the same keyword and document pair, we set one of these to the correct term frequency and make the remaining term frequencies zero. These latter postings are now essentially padding to mask whether a new term was added to the document. After processing an update with m postings, we have that `inv`'s size increases by exactly $(W+1)\cdot m$ bytes. Stage ④ in Figure 3 shows the primary index merged with updates on both existing and new keyword and document pairs.

Merging large collections of outstanding updates can degrade search performance for update-heavy workloads. We can easily add to clients the ability to perform auto-merge updates, i.e., when the number of outstanding updates passes some threshold (in terms of total number of postings) an update additionally triggers downloading the current index to merge outstanding updates. This does not affect security, and we report on the benefits of auto-merging in the experiments section.

5.4 Vertical Partitioning

While providing stronger security than previous encrypted search systems, the full-download construction may result in impractical bandwidth for cold-start searches over large document sets. Here we introduce a performance optimization: vertical partitioning. The high-level idea is to split an index so that posting lists are spread across multiple different encrypted indexes. When searching for a keyword, one fetches only the first partition to obtain the first page (or more) of results. If the user desires more results, the client can fetch subsequent vertical partitions. Compared to the full-download construction, the only new leakage introduced by vertical partitioning is the number of pages requested by a user. This enables significantly better bandwidth usage in the common case where only the first page of results are needed.

Our vertical partitioning strategy works by splitting the index across some number of *levels*. Each level has associated to it an encrypted index $P_1,\ldots,P_L$ and an encrypted update cache $\vec{U}_1,\ldots,\vec{U}_L$, both encoded and encrypted before being stored at the server. The updates will still be merged in opportunistically, but now according to a more complicated schedule, as only part of the index will typically be available for merging at any given time.

Leveled size-locking. In order to describe our approach abstractly, we generalize the notion of size-locking from the prior section in a manner that fits our application. Our approach will use three algorithms:

- A stateful algorithm SLEncodeUp that takes as input an update Δ and emits a bytestring, along with updated state.
- A stateful algorithm VMerge that take as input a bytestring P encoding the current index and a vector of bytestrings $\vec{U}$ encoding some information to be merged. It outputs a new bytestring P encoding the updated index, along with a bytestring U encoding the "evicted" information. It updates its state (denoted $\mathsf{st}_{\mathsf{mrg}}$ in our algorithms) on each run.
- An algorithm RunQuery that takes as input one or more

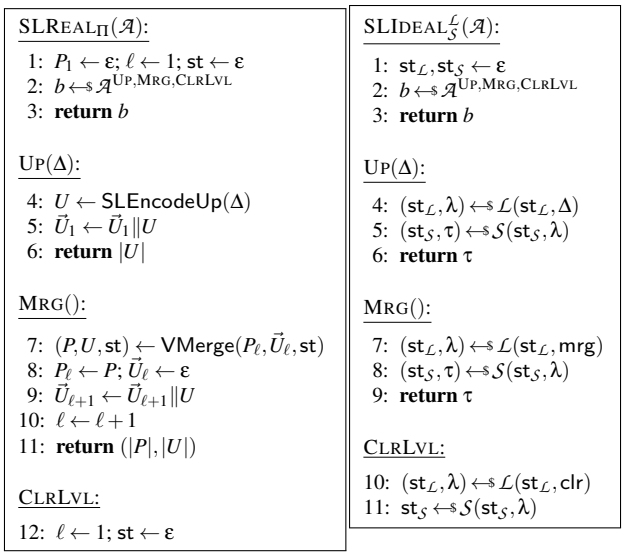

Figure 5: Security games for generalized size-locking.

bytestrings $P_1, \ldots, P_\ell$ encoding index information along with a query q. It returns a result set G.

Our size-locking requirement is now more complicated and is expressed via a real-versus-ideal (perfect) simulation requirement. Intuitively, the real game maintains a set of encoded indexes $P_1, P_2, \ldots$, and associated update caches $\vec{U}_1, \vec{U}_2, \ldots$. The adversary can ask for updates to be encoded and appended to the top cache $\vec{U}_1$, after which it can observe the size of the encoding. The adversary can also for merges to occur, in a restricted top-to-bottom manner. Upon merge request, the "current level" (tracked by variable ℓ) is merged with its update cache. After this, P_ℓ is overwritten, its cache $\vec{U}_\ell$ is cleared, and the "evicted" information U is appended to the next level cache. Finally, ℓ is incremented, and the sizes of the new index and the evicted information are returned.

The following definition asks that the real game can be perfectly simulated using only the leakage from a stateful algorithm $\mathcal{L}$.

Definition 7. *Let* $\Pi = (\mathsf{SLEncodeUp}, \mathsf{VMerge})$ *be pair of algorithms fitting the syntax above. Let* $\mathcal{L}$ *be an algorithm. We say that* Π *is* $\mathcal{L}$*-leveled-size-locked if there exits an algorithm* $\mathcal{S}$ *such that for all* $\mathcal{A}$*,*

$$Pr[\text{SLREAL}_\Pi(\mathcal{A}) \Rightarrow 1] = \Pr[\text{SLIDEAL}^{\mathcal{L}}_{\mathcal{S}}(\mathcal{A}) \Rightarrow 1].$$

Our leveled construction. We now describe how search and updates work for vertical partitioning. Figure 6 gives pseudocode, which uses SLEncodeUp, VMerge and RunQuery that we detail afterwards.

The server memory is structured as a sequence of encoded and encrypted indexes $B_1, B_2, \ldots$ each with an associated vector of update caches $\vec{C}_1, \vec{C}_2, \ldots$. The client state st always consists of some locally-stored copies of encoded level indexes $P_1, \ldots P_\ell$ encrypted in $B_1, \ldots, B_\ell$, some cached updates $\vec{U}$ that reflect the updates in the-level cache, and the state $\mathsf{st}_{\mathsf{mrg}}$ used by VMerge.

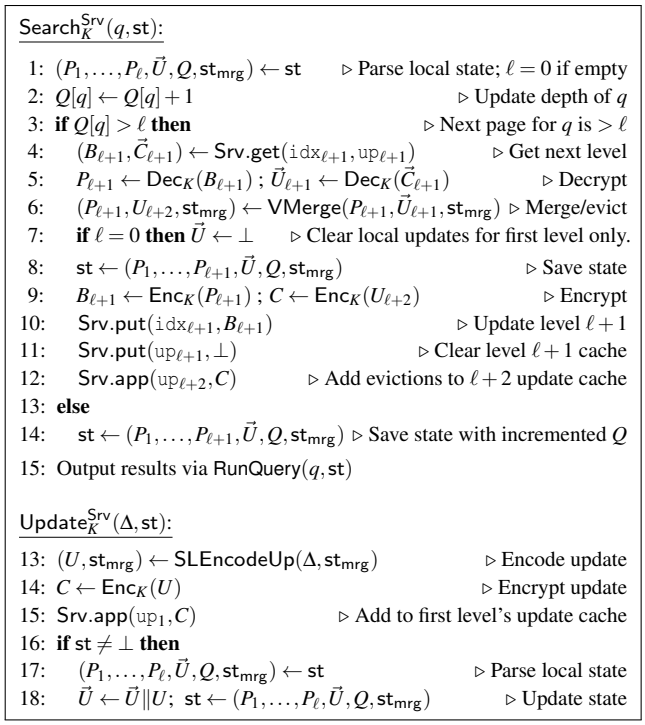

Figure 6: Our vertically-partitioned encrypted index construction.

The top-level cache will be used for lazy updates similar to before (see lines 13-15 of Update). The client then encrypts the updates, appends it to the first level update cache, and updates the $\vec{U}$ in its state, if there is any. Note that the local variable $\vec{U}$ follows the data stored at the server as $\mathtt{up}_1$ (but unencrypted).

We walk through how search begins from a cold-start, i.e. with $\mathsf{st} = \perp$. Searching begins by downloading just the first level $(B_1, \vec{C}_1)$, decrypting and decoding them (lines 3–4). Next, algorithm VMerge merges the encoded first-level index P_1 and the updates $\vec{U}_1$, emitting a new copy of $\vec{U}_1$, some evicted data U_2. Then, as a special case for the first level, the local changes $\vec{U}$ can be deleted (line 6). (Future searches will need to retain the updates that have happened since a cold-start search.) Finally, the new level P_1 is added to the state, and then encrypted and to the server to overwrite the existing second level index. The second level cache is cleared, and the evicted information is appended to the second level's cache. Finally, search is performed using the locally-held indexes and update cache.

A subsequent invocation of search with the same query will trigger a downloading of the next level (the required level is tracked in a table Q of per-query counters). In this case we fetch, decrypt, decode, and then merge the information from the next level into the local state as before. The only difference is that the local-held updates $\vec{U}$ must be retained while merging lower levels, since they will not be merged into the first level until a cold-start search.

Merging and evicting. All that remains to complete our construction is an instantiation of SLEncodeUp, VMerge, and

RunQuery. The first algorithm is the same as before (except that it maintains some state, as noted below), and the third is straightforward once we have described the data format. We describe VMerge in two stages: First we discuss how it decides what to merge versus evict, and then we describe the low-level encoding formats, which are extensions of our earlier encodings.

Algorithms SLEncodeUp and VMerge maintain a state consisting of N, the total number of postings added to the system and ℓ, the depth of the last level downloaded, and a table recording for each keyword how many of its postings are stored in level up to ℓ (this table can be inferred from the local state, but it is useful conceptually). When presented with an encoded index P and update cache $\vec{U}$, VMerge will determine a pinned size for the index, and pack in postings for keywords as evenly as possible, breaking ties arbitrarily, and will evict the rest if they do not all fit (which is the eventual steady-state for each level).

We would like each level to minimally include k postings for each keyword, to ensure that the first k results for every (single keyword) query are available given just the first level. To do this securely, we pin the number of postings in each level to a function of N, the total number of postings, and attempt to fit in all of the relevant postings as follows. First, fix an ordering over keywords that matches the ordering in which they were added to the corpus, and break ties (due to being added by the same document) arbitrarily. Then we loop over keywords in that order, adding to their posting list the next highest posting by BM25 for that keyword. BM25 ties within a list are broken arbitrarily. This round robin approach ends when the pinned size is reached or all postings have been processed. To get subsequent levels' postings, remove all the postings from the first level from consideration, and otherwise repeat the process (with a different α, as described below).

Size-locked encoding details: First level. The first level uses a different encoding format from later levels. Updates at this level are managed exactly as before. For the index P_1, the encoding is mostly the same as our full-download construction except for two differences: First, we pin the size of the look-up table `bin` at $\mathsf{Bin}_{\alpha_1}(N)$, where α_1 is a system parameter fixed at setup time. Looking ahead, we choose α_1 to ensure this first-level table includes entries for every keyword.

The second difference is that we change the encoding of `inv`. Because we will not be storing all the postings, we must encode each term's *document frequency* to enable BM25 computation. Previously these were omitted since they could be recomputed by summing the term frequencies. Just adding a fixed-length encoding of each keyword's document frequency would leak the number of keywords. Instead, we process the posting list for keyword w to compute its document frequency df, and encode it in as many bytes as needed, in little-endian order; call these bytes $\mathsf{df}_1, \mathsf{df}_2, \ldots$, where the unused bytes are implicitly zero. The posting list for w is encoded as

$$(H(w) \vee 10^{8W-1}) \,\|\, \widetilde{\mathsf{tf}}_1 \,\|\, \mathsf{df}_1 \,\|\, \mathsf{id}_2 \,\|\, \widetilde{\mathsf{tf}}_2 \,\|\, \mathsf{df}_2 \,\|\, \cdots \,\|\, \mathsf{id}_\ell \,\|\, \widetilde{\mathsf{tf}}_\ell \,\|\, \mathsf{df}_\ell.$$

Note that while the document frequency byte encoding varies in length, it will not be leaked, because we pad each posting regardless. We will have enough space for the df encoding, because a list of length ℓ gives ℓ bytes to encode[1] $\mathsf{df} = \ell$.

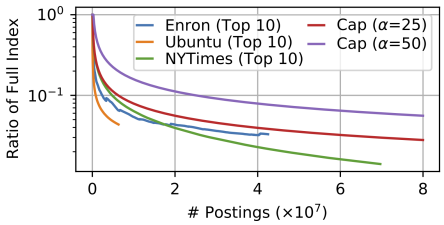

Figure 7: Comparing $\mathsf{Cap}_{\alpha,k}(N)$ to N_{10}, the total number of postings in the top-10 of postings-lists across three datasets. We added documents in random order, measuring the ratio N_{10}/N after each addition. For the Cap curves we plot $\mathsf{Cap}_{\alpha,k}(N)/N$ with $k = 10$. Heaps' Law is apparent in the shape of each curve, and our choice of Cap proves conservative.

We next describe how the size of the top-level index is pinned. For this use the capacity function

$$\mathsf{Cap}_{\alpha_P,k}(N) = \min\{N, k \cdot \mathsf{Heaps}_{\alpha_P,0.5}(N)\} = \min\{N, k\alpha_P N^{0.5}\},$$

where Heaps is the function from Heaps' law (used previously in Section 5.1 and above in sizing `bin`), k is the number of postings we desire in the first page of results, and α_P is another system-wide parameter chosen at setup time. The intuition is that the corpus will typically have $\mathsf{Heaps}_{\alpha_P,0.5}(N)$ unique terms (Heaps' law with $\beta = 0.5$), and thus $k \cdot \mathsf{Heaps}_{\alpha_P,0.5}(N)$ postings are typically sufficient in each level. In Figure 7, we compare $\mathsf{Cap}_{\alpha_P,k}(N)$ to the actual number of postings in the top-k of any posting list, for $k = 10$ and the datasets we use for evaluation in Section 6. As can be seen, using $\alpha_P = 25$ provides a conservative overestimate that suffices in practice; estimation error only affects performance and not security. From now on we let $\alpha_P = 25$.

For an index with W-byte identifiers, M-byte metadata, n documents, and N total postings, the first-level index encoding will have length

$$(2W+8) \cdot \mathsf{Bin}_{\alpha_1}(N) + \left(\frac{3W}{2} + M\right) \cdot n + (W+2) \cdot \mathsf{Cap}_{\alpha_P,k}(N).$$

Size-locked encodings for other levels. We now describe the encoding of indexes P_i for levels $i > 1$. The input for each level is a partial posting list of the form

$$H(w),\ (\mathsf{id}_1, \widetilde{\mathsf{tf}}_1),\ \ldots,\ (\mathsf{id}_\ell, \widetilde{\mathsf{tf}}_\ell).$$

where the posting lists are lexicographically ordered by the pair consisting of their document frequency and word hash (which will be available because we will only operate on higher level indexes after loading the first level). It then encodes this a posting list as

$$(\mathsf{id}_1 \vee 10^{8W-1}) \,\|\, \widetilde{\mathsf{tf}}_1 \,\|\, \mathsf{id}_2 \,\|\, \widetilde{\mathsf{tf}}_2 \,\|\, \cdots \,\|\, \mathsf{id}_\ell \,\|\, \widetilde{\mathsf{tf}}_\ell.$$

[1] Formally, we use that $256^\ell - 1 > \ell$ for all positive integers ℓ.

In words, the first identifier has its top bit set, and the rest are encoded with their rounded term frequencies (and top bits cleared) exactly as before. This encoding does not include the hashed terms themselves. The decoder can infer this association using the document frequencies from the top partition (the first posting list corresponds to the term with highest document frequency/word hash pair and so on). We include at level $i > 1$ exactly $\mathsf{Cap}_{\alpha_i,k}(N)$ postings, where α_i is a parameter. Looking ahead, we will make $\alpha_i < \alpha_1$ for all $i > 1$ since fewer keywords have more than k postings. In addition, the document identifier is not needed for level $i > 1$. Hence, the byte-length of the encoding of P_i is $(W+8) \cdot \min\{\mathsf{Heaps}_{\alpha_i,0.5}(N), N_i\} + (W+1) \cdot \mathsf{Cap}_{\alpha_P,k}(N)$, where N_i is the number of postings in P_i. Note that the first term accounts for the size of `bin` for P_i. The last partition may contain fewer than $\mathsf{Cap}_{\alpha_P,k}(N)$ postings, in which case the length is appropriately adjusted — its length can at most reveal N.

The evictions are encoded in the same format as the size-locked index encoding at level $\ell+1$, i.e., the `bin` with $\min\{\mathsf{Heaps}_{\alpha_i,0.5}(N), E\}$ terms (where E is the number of evicted postings), and an `inv` that contains all the evicted postings. Note that `bin` in the eviction encoding is used to locate the posting list bytestrings of each word in the evictions. The size of the evictions is:

$$(W+8) \cdot \min\{\mathsf{Heaps}_{\alpha_i,0.5}(N), E\} + (W+1) \cdot E$$

These evictions will be merged into the level $\ell+1$ index the next time it is fetched.

Size-locking analysis. The next claim formalizes the size-locking property of our encoding. Intuitively, for updates all that is leaked is the number of postings and whether they contain new documents. For merging only the level being merged is leaked. Formally, the claim uses the following leakage function $\mathcal{L}$: It maintains as state a set of identifiers S and a counter ℓ. On input an update Δ, it outputs the number of postings in Δ and a bit indicating if the associated document identifier is in S, and it updates the state by adding the identifier to S. On input mrg, it outputs ℓ from its state and then increments ℓ. On input clr, it sets $\ell \leftarrow 1$ and has no output.

Claim 8. *Let* $\Pi = (\mathsf{SLEncodeUp}, \mathsf{VMerge})$ *and* $\mathcal{L}$ *be as described above. Then* Π *is* $\mathcal{L}$*-sized locked.*

Putting everything together. A formal analysis of the resulting encrypted index construction is given in the full version. In addition to the information leaked by FULL, this construction also reveals the vertical partition "depth" of a query, which does not appear to enable any practical and damaging attacks.

5.5 Horizontal Partitioning

Our next extension is simple: we just horizontally partition the keyword space and build separate (vertically partitioned) size-locked indexes for each subset of the keyword space. We assign each term to one of P buckets via a pseudorandom function (PRF), and run P parallel versions of either our vertically-partitioned construction. This reduces the search bandwidth by approximately a factor of P (for single-term queries), at the cost of extra leakage (because touching the buckets limits the possibilities for the query or update). Techniques similar to horizontal partitioning have been studied in the information retrieval literature [32, 47] for load balancing and parallel index lookup, and also recently for DSSE schemes [16].

We implement this by having the user derive an additional key K' for a PRF (e.g., HMAC-SHA256), and assigning a term to a bucket via $p = \mathrm{PRF}(K', w) \bmod P$. For updates, the new postings are assigned to their respective partitions, and an update is issued to each partition with the assigned postings. A query for a single keyword is run using one partition, and a multi-keyword query is run by executing the query against the relevant partitions and then ranking the results together (once the partitions are downloaded, we can compute BM25 on the entire query). We only take the first vertical level for each involved partition, and if further results are requested, then we fetch the subsequent levels for all keywords one at a time as needed. In a search session some levels of some partitions may be cached locally, in which case we only fetch the omitted ones.

The expected number of words in each partition is $\frac{1}{P}$ of the total number of words, and so for vertical partitioning we divide the estimated number of words from Heap's Law over the entire document collection by P to determine the sizes of `bin` and `fwd`. Otherwise their construction is the same.

5.6 Progressive Partitioning

The amount of partitioning to use provides a tunable tradeoff between security and efficiency. Vertical partitioning has more leakage than FULL by revealing level accesses. Search with horizontal partitioning leaks which partitions are accessed, allowing some frequency information at the granularity of the number of partitions.

We suggest that for practical deployment, a search system should use a *progressive partitioning* strategy. The idea is simple: conservatively use the most secure approach (full-download) for as long as possible, and only when it becomes too unwieldy in terms of performance does one start to use partitioning. This ensures that users by default get full security, and only trade-off security for performance when it is necessary. It also means that, from a leakage perspective, what is revealed may be less damaging given the size of the dataset when leakage starts being revealed (e.g., each horizontal partition will have large number of keywords associated to it).

A security-relevant subtlety is that the decision to progress to a new partitioning strategy must be based on leakable information. We take an expedient approach. Let $N_1 < N_2$ denote

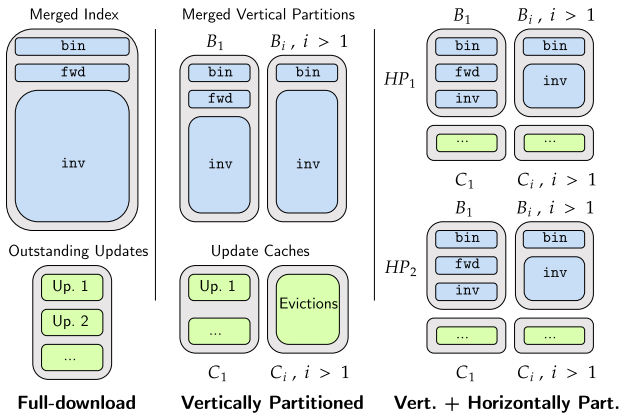

Figure 8: The three stages of our progressive construction.

two thresholds, and we add to our indexes an encoding of the number of postings added so far. Then after N_1 postings have been added, the client will progress to use vertical partitioning. This requires a one-time operation to refactor the index and upload the newly partitioned encrypted index. Later, when N_2 postings have been added, the client progresses to horizontal partitioning. This requires downloading all vertical levels, and factoring it into a fixed number of horizontal partitions (e.g., 10 or 100). Figure 8 shows a visual depiction of the layout of indexes for the three different settings.

6 Experimental Evaluation

Our experimental analysis addresses the following questions: (1) Do our constructions provide accurate ranked search results? (2) How practical are our encrypted index constructions in terms of bandwidth and end-to-end operation time? (3) How do they compare to DSSE type constructions?

Implementation. We implemented our construction in Python, and we plan to release our code as a public, open-source project. Our prototype implementation currently only supports whole document additions, which is sufficient to handle our experimental workloads. Adding/deleting words to/from existing documents (via always adding postings, sometimes as dummies) is a straightforward extension, and our design is such that these operations' performance will be strictly better than adding a new document (assuming the same number of postings modified versus added).

We ran our experiments using PyPy3. Our code uses the PyCryptodome library's implementation of AES-GCM-128 for authenticated encryption, and HMAC-SHA256 for a PRF. For all symmetric cryptographic tools, we fix the key size to be 128 bits. We used BLAKE2b to compute word hashes. We fix the document metadata size at $M = 6$ bytes (4 bytes for the name; 2 bytes for encoding the number of words in the document).

It will be useful for reporting results to focus on our construction operating with different combinations of optimizations: FULL (Section 5.3) has no partitioning, VPART (Section 5.4) uses vertical partitioning, VHPART (Section 5.5) includes both vertical and horizontal partitions, and PROGRESSIVE (Section 5.6) refers to the complete construction that progressively partitions according to configurable thresholds. These constructions are heavily parameterized, and we summarize these parameters in Figure 9 along with the default values we use unless specified otherwise.

Parameter	Description	Default
W	hash width (bytes)	4
M	document metadata (bytes)	6
P	number of horizontal partitions	10
a	auto-merging threshold (num. postings)	40,000
α_P	Heaps α for vert. part. size	25
α_1	Heaps α for lookup table in vert. part. 1 and FULL	90
α_i	Heaps α for lookup table in vert. part. i, $i > 1$	20
β	Heaps β	0.5
N_1	FULL to VPART threshold (num. postings)	1×10^6
N_2	VPART to VHPART threshold (num. postings)	2×10^6
k	page size (num. results)	10

Figure 9: Our construction's configuration parameters and the values we focus on (unless specified otherwise).

Experimental setup. We ran experiments in a networked setting with Redis, a high-performance in-memory key-value store, as the storage service provider. We use AWS EC2 t2.large instance type in *US East 2* with Intel dual-core 2.3GHz CPU and 8 GB RAM as a client. For Redis, we used one *Standard 1 GB* Azure Redis cache at *US East*, with default configurations. For some of our experiments with a DSSE scheme, we had to upgrade to a larger Redis instance type. To parallelize experiments, we used separate EC2/Redis instance pairs for each dataset and used separate pairs for the Enron microbenchmarks and macrobenchmarks.

The reported maximum read and write bandwidths from the Azure portal for one of our instance pairs were 127 Mbps and 81 Mbps, respectively, and the average round-trip latency from the EC2 instance to the Azure Redis service was 13 ms. These were representative of all instances used. Below, when we report on bandwidth this is as measured at the application layer (excluding standard TCP/IP headers).

Datasets. We ran our experiments using the Enron email collection (Enron) [2], the Ubuntu IRC dialogue corpus (Ubuntu) [4], and a collection of New York Times news articles (NYTimes) [3]. These are all public datasets that have been used in prior work on searchable encryption [10, 21, 48] and information retrieval [29, 44]. In this section, we focus on the Enron dataset: it includes 517,310 documents, 338,913 keywords, and 42,510,783 postings (after stop-word removal [30] and Porter stemming [37]). We provide detailed statistics on the three datasets, as well as experimental results on the Ubuntu and NYTimes datasets in the full version.

Search quality. We start with search quality. Recall that our constructions provide approximate BM25 scoring, and also have occasional hash collisions which can degrade search quality. More broadly we can evaluate search quality by com-

	# Postings	Bandwidth (MB) 10% Enron 4.3×10^6	50% Enron 21.3×10^6	100% Enron 42.5×10^6	Overall Latency (sec) 10% Enron 4.3×10^6	50% Enron 21.3×10^6	100% Enron 42.5×10^6
Query	FULL	25.09	116.30	228.15	1.61 (±0.32)	6.80 (±1.03)	12.80 (±2.23)
	VPART	6.72	16.68	25.38	0.78 (±0.24)	1.69 (±0.36)	2.47 (±0.39)
	VHPART-10	1.17	4.16	7.51	0.16 (±0.06)	0.33 (±0.03)	0.46 (±0.06)
	VHPART-100	0.34	1.52	2.96	0.09 (±0.01)	0.29 (±0.18)	0.43 (±0.17)
	CTR-DSSE	1.48	4.35	−−	1.71 (±1.08)	4.83 (±4.16)	−−
Query w/ Merge	FULL	51.88	237.18	463.33	4.76 (±0.48)	19.93 (±1.12)	48.38 (±1.85)
	VPART	13.88	33.91	51.37	2.87 (±0.25)	6.11 (±0.29)	9.22 (±0.70)
	VHPART-10	2.50	8.56	15.33	0.81 (±0.12)	1.69 (±0.18)	2.52 (±0.14)
	VHPART-100	0.72	3.11	6.02	0.40 (±0.03)	0.86 (±0.18)	1.18 (±0.14)

Figure 10: Bandwidth in megabytes (10^6 bytes) **(left)** and average time in seconds **(right)** for search queries as well as search queries that additionally merge outstanding updates. Query times are averaged over 30 executions; standard deviations are given in parentheses.

paring to a baseline of using BM25 [39] for plaintext search. For a query q whose result is an ordered list of documents $\mathcal{R}$, we measure the overall search quality using *normalized discounted cumulative gain* (NDCG) [22, 30], which aggregates the scores of the results, with more weights on the earlier ones, i.e.,

$$\mathsf{NDCG}(q,\mathcal{R}) = \frac{1}{\mathsf{IDCG}(q,|\mathcal{R}|)} \sum_{i=1}^{|\mathcal{R}|} \frac{2^{\mathsf{BM25}(q,\mathcal{R}_i)}-1}{\log(i+1)},$$

where $\mathsf{IDCG}(q,k)$ is a normalization factor, calculated from the optimal ranking of the top-$|\mathcal{R}|$ results for q, to make $\mathsf{NDCG}(q,\mathcal{R}) \in [0,1]$. Higher NDCG indicates better search quality, relative to BM25 plaintext search. Averaging the NDCG's of multiple queries gives a measure of search quality.

Considering just the top 10 results, the NDCG over 50 random single-keyword searches for all our techniques never drops below 0.9985. This means we match the search quality of state-of-the-art plaintext search systems for the first page of results. We also measured the keyword hash collision ratio, which is calculated as $\frac{\#\ unique\ keywords - \#\ unique\ word\ hashes}{\#\ unique\ keywords}$ for the three datasets. It is always less than 10^{-4}, and hash collisions never impacted our search quality evaluations because the probability of randomly picking these keywords is small.

Performance. We evaluate the performance of our constructions using all three datasets. Due to space constraints, we defer most of the details to the full version and focus here on a subset of the results for Enron that highlight key points.

In terms of size, the FULL index is 228.15 MB for the full Enron dataset. Using vertical partitioning reduces by an order of magnitude, to 25.38 MB, the size of the first level index which suffices for the first page of results. These sizes directly impact query performance, as seen in Figure 10 (top group of rows): in our experimental setup performing a single cold-start search with FULL for the entire corpus took 12.80 seconds. Here we report on the average time over 30 single-keyword queries; the keywords were chosen to cover a wide range of document frequencies (see the full version). Vertical partitioning cuts query time down to a couple seconds for the first page of results, and just 10 horizontal partitions gives practical search at 460 ms. Should a user request more results, subsequent levels are even smaller than the first level, and search times are likewise faster than for the initial level, assuming the subsequent level is merged. Subsequent searches (on other keywords) in the same session can be handled locally, which is trivially fast and a key advantage of our approach over others (such as DSSE, see below).

Lazy updates are fast — inserting 100 random Enron documents takes as little as 0.21 seconds for FULL, and as much as 3.39 seconds for VHPART-100. Updates for VHPART-100 are more expensive because most updates require updating many partitions. The operations that trigger merges are more expensive: either a search that has outstanding updates or an auto-merge (which has a similar performance profile). The second group of rows in Figure 10 gives the total time to perform a search on an initialized index that has the updates for 100 inserted documents outstanding. Bandwidth used is a bit more than 2× that of a regular query as we not only have to download the encrypted index and updates, but upload the newly merged index. Consequentially, the time to complete the full operation is also larger, in the worst case with FULL this takes up to roughly 50 seconds, which may be too slow in some contexts. We note here that the outstanding updates for each query are accumulated across queries in these experiments. Partitioning reduces costs significantly. We emphasize that these times are not the user-visible latency, as the merging and upload can happen in the background.

Macrobenchmarks. We evaluate the performance of the PROGRESSIVE construction using synthetic workloads of update (U) and query (Q) operations, and compare with two baselines. We conduct this evaluation in a cold-start setting — the client always downloads the index and any outstanding updates from the server to answer a query. As mentioned above, warm-start searches are handled locally and are therefore trivially fast.

To generate workloads, we implemented a program that, given an input dataset, outputs a transcript of update and query operations. The total number of operations and the desired ratio of updates to queries are configurable. The workload generation proceeds by randomly determining each operation as an update or a query, according to the desired ratio. An update operation is an addition to the index of a new document chosen uniformly from the dataset (without replacement).

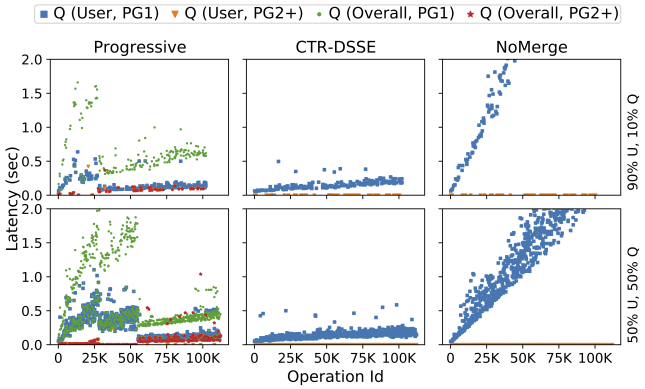

Figure 11: Macro-benchmark results for two types of workloads, i.e., *update-heavy* with 90% updates (U) and 10% queries (Q) in the top row, and *balanced* with 50% updates and 50% queries in the bottom row. Outliers are not shown but are explained in the text.

A query operation consists of a single keyword that is uniformly sampled from the currently indexed keywords,[2] as well as a number indicating the number of pages of results needed to retrieve all of the postings for this query (given the current state of the index). When vertical partitioning is in-use, if the number of pages for the query is k, we split the query operation into at most k queries, one for each level, and evaluate the performance of fetching each level separately while assuming states downloaded for levels $1,\ldots,i-1$ are available at the client for level i. Depending on the state of the index during execution, the actual number of fetches from the storage service may be lower than k should the search query be answerable with the already fetched portion of the index. Therefore, a single requested query operation may expand into multiple performed operations, one for each page of results. Consequently, although the number of requested operations in the workload shown in Figure 11 is 100,000, the number of performed operations exceeds 100,000.

We experiment with two workloads: (1) an *update-heavy* workload of 100,000 operations, with 90% updates and 10% queries; (2) a *balanced* workload of 100,000 operations, with 50% updates and 50% queries. The performance for PROGRESSIVE using the update-heavy (top) and balanced synthetic workloads (bottom) are shown in Figure 11 (left-most two charts). First, with the update-heavy workload (top), we can see that PROGRESSIVE switches from FULL to VPART at around operation 10,000, and then from VPART to VHPART-10 at around operation 25,000. Overall query time increases rapidly, but note that the majority of this time is merging in outstanding updates and re-uploading the index. The user-visible lag time for search (blue squares and orange triangles) remain under one second for the vast majority of operations. We note that the number of keywords among the 10 horizontal partitions when switching from VPART to VHPART is 4,995 ($SD = 78$), which implies a reasonable level of uncertainty for leakage abuse adversaries that perform frequency analysis on partition accesses (see the full version).

For the balanced workload (bottom), the progressions happen later than those for the update-heavy one (a bit after operation 25,000 and after operation 50,000), but otherwise the trends are consistent with the update-heavy workload.

We down-sampled the plots to make them easier to read, and due to this, outliers are not shown. For completeness, we summarize them here. The update-heavy macrobenchmark consisted of 12,371 queries. Of these, 31 had overall latencies greater than 2 seconds and 9 had latencies greater than 5 seconds. The balanced macrobenchmark consisted of 62,069 queries. Among them were 320 queries with overall latencies greater than 2 seconds and 45 queries with latencies greater than 5 seconds. The largest outlier was a 64.5 second search query with a user-perceived latency of 8.9 seconds.

We note that these larger outliers are due either to shifts from one index format to another or expensive merges on outstanding updates. An additional form of auto-merging we employ in vertically partitioned constructions in our macrobenchmarks involves merging the evictions in the next vertical partition of a query, if the size of the encodings in its update cache surpasses a certain threshold, before sending it additional evictions. In our experiments, we used 2^{21} bytes as the threshold for the VPART phase and 2^{18} bytes as the threshold for the VHPART phase. Large outliers can be mitigated with improved auto-merging policies, which we leave to future work.

Comparison with other approaches. We also compare our construction against two baseline approaches. The first is a simplification of our size-locked approach that keeps appending updates, without merging them at all. We call this the NOMERGE construction. To perform an update, the client appends an update to the outsourced storage. To perform a search, the client downloads and processes all the updates. We optimized searches by performing them in a streamed manner using Redis, which does not suffer from the fragmentation-related performance issues we observed with other storage services. One can further optimize the construction by horizontally partitioning the keyword space, and doing so leaks a subset of VHPART's leakage: horizontal partition leakage but no vertical partition leakage. Here we use 30 partitions, which roughly matches the per-partition size of the first level of VHPART-10.

This straw proposal for a scheme actually performs well for very small indexes (rightmost charts in Figure 11), but user-visible performance to obtain the first page of results (blue squares) quickly becomes prohibitive given the linear growth with updates, even with partitioning across thirty partitions. We observe a linear growth in search latency for NOMERGE because each appended update requires independent decryption and processing before a search can be handled, and so the overall cost increases as the number of updates increases. Even though NOMERGE is simpler with comparable security

[2] Search queries on keywords not in the current index perform even better.

to VHPART, its performance degradation after many updates makes it much less practical than PROGRESSIVE.

The second baseline is a DSSE-based construction called CTR-DSSE. It extends an existing forward-private DSSE construction [9] to handle lightweight clients and ranking, while preserving the underlying leakage profiles. We detail its construction and security analysis in the full version. As expected, this performs well (Figure 11 middle charts), particularly for these workloads that tend to have small posting lists: the majority of keywords have small postings in this dataset and so the average posting length is small in our macrobenchmarks. We note that the performance of CTR-DSSE degrades when searches are conducted on keywords with more postings. Our microbenchmarks on search highlight this, as they are sampled to have a variety of posting list lengths and there we see higher average query latencies for CTR-DSSE (Figure 10). There we omit 100% Enron for CTR-DSSE in the microbenchmarks because the resulting indexes caused the Azure Redis instance to max out on memory.

PROGRESSIVE achieves performance competitive with that of CTR-DSSE. This is especially the case when we examine the user-perceived latencies — the time between when a user performs a query and sees the query results.

Remarks. The server-held encrypted indexes in our schemes grow monotonically with the number of updates, even in the case that the documents do not grow in size. For example, a large sequence of modifications to a file will increase the size of the index but not the file. In our vertically-partitioned solutions, this does not affect expected bandwidth usage, as the repeated postings due to updates can be evicted to lower levels that are less likely to be downloaded. In some applications, like outsourced document editing, documents already grow monotonically in order to maintain histories, and so additional storage overhead of a growing index may be tolerable.

Extending our constructions to limit growth over time appears to be difficult. One potential approach, periodically performing garbage collection on indexes to remove duplicate postings from the merged index, would seem to lead to the exact sort of leakage we aim to avoid. Whether one can construct size-locked indexes that do not monotonically increase in size remains an open question.

7 Related Work

The study of keyword search for encrypted data began with work by Song, Wagner, and Perrig [40] (SWP). They propose schemes for linear search of encrypted files, and mention that their approach can be applied instead to reverse indices that store a list of document identifiers for each keyword. Subsequent work by Kamara et al. [14] provided formal, simulation-based security notions and new constructions that leak less information than SWP's approaches. Cash et al. [11, 12] introduced some of the simplest known SSE schemes, by leveraging state on the client side (beyond the key).

SSE schemes do not provide all the features that are standard in plaintext search systems. The schemes mentioned so far only support single keyword searches, but follow-up works have suggested schemes that support boolean queries [12, 23] at the cost of additional leakage. Traditional SSE schemes do not support relevance ranking, pagination, or previews. Baldimitsi and Ohrimenko [5] give a scheme that supports result ranking, but assume non-colluding servers and do not support updates.

Dynamic SSE (DSSE) schemes [7–9, 11, 13–15, 17, 24–27, 31, 31, 40, 41, 46] do support updates, including deletions, but none we know of support ranking of queries. Amongst these, the Blind Storage system of Naveed, Prabhakaran and Gunter [34] is notable in that it operates with constant client state and a storage-only server, like our construction. A DSSE approach by R. Lai and Chow [28] does not explicitly discuss ranking or metadata previews, but can be extended to do so at the cost of extra leakage.

As mentioned in Section 2, existing, efficient SSE constructions are vulnerable to two classes of attacks: leakage-abuse attacks (LAAs) [6, 10, 18, 19, 21, 33, 38, 43, 45, 48] and injection attacks [6, 10, 48]. Some prior works have focused on hiding the sizes of data in order to reduce leakage. Kamara and Moataz [24] and Patel, Persiano, Yeo and Yung [35] construct encrypted multimaps that hide the number of results returned while minimizing redundant padding, disrupting some leakage-abuse attacks. A recent work of Demertzis, Papadopoulos, Papamanthou, and Shintre [16] presents SEAL, a *static* encrypted database construction that combines ORAM and padding to provide adjustable leakage. We note that their use of ORAM over parts of their database is similar to our horizontal partitioning. SWiSSSE [20] reduces leakage through a randomized client-to-server write-back strategy and keyword frequency bucketization and padding. However, SWiSSSE doesn't explicitly support document ranking, nor is it clear if it prevents injection attacks.

A line of work on forward-private DSSE [8, 9, 17, 26, 41] seeks to partially mitigate injection attacks by ensuring that past searches cannot be applied to newly added files. But this doesn't prevent injection attacks because they still work on all future queries. Applying expensive primitives, e.g., ORAM [16, 42], on the encrypted index can prevent injection attacks based on the result pattern and, if one additionally uses padding, injection attacks based on response size.

In summary, no prior DSSE scheme provides ranking or works with stateless clients, and they all have strictly worse security than our FULL construction. The leakage of the partitioned constructions VPART and VHPART is formally incomparable to prior work, but still resists injection attacks. The primary limitations of our constructions are high bandwidth and the lack of a mechanism to reclaim space upon deletions.

Acknowledgements. This work was supported in part by NSF CNS grants 1703953 and 1704296.

References

[1] Apache lucenetm 7.7.3 documentation. https://lucene.apache.org/core/7_7_3/index.html.

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

[3] New york times news. https://archive.ics.uci.edu/ml/datasets/Bag+of+Words.

[4] Ubuntu dialogue corpus. https://www.kaggle.com/rtatman/ubuntu-dialogue-corpus.

[5] F. Baldimtsi and O. Ohrimenko. Sorting and searching behind the curtain. In *FC 2015*, volume 8975 of *LNCS*, pages 127–146.

[6] L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. In *Poceedings of the 27th Network and Distributed System Security Symposium, NDSS'20*, 2020.

[7] J. Blömer and N. Löken. Dynamic searchable encryption with access control. In A. Benzekri, M. Barbeau, G. Gong, R. Laborde, and J. Garcia-Alfaro, editors, *Foundations and Practice of Security*, pages 308–324, 2020.

[8] R. Bost. $\Sigma o \phi o \varsigma$: Forward secure searchable encryption. In *ACM CCS 2016*, pages 1143–1154, 2016.

[9] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *ACM CCS 2017*, pages 1465–1482, 2017.

[10] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM CCS 2015*, pages 668–679, 2015.

[11] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS 2014*, 2014.

[12] 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 *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 353–373, 2013.

[13] J. G. Chamani, D. Papadopoulos, C. Papamanthou, and R. Jalili. New constructions for forward and backward private symmetric searchable encryption. In *ACM CCS 2018*, pages 1038–1055, 2018.

[14] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS 2006*, pages 79–88, 2006.

[15] I. Demertzis, J. G. Chamani, D. Papadopoulos, and C. Papamanthou. Dynamic searchable encryption with small client storage. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.

[16] I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre. SEAL: Attack mitigation for encrypted databases via adjustable leakage. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[17] M. Etemad, A. Küpçü, C. Papamanthou, and D. Evans. Efficient dynamic searchable encryption with forward privacy. *PoPETs*, 2018(1):5–20, 2018.

[18] M. Giraud., A. Anzala-Yamajako., O. Bernard., and P. Lafourcade. Practical passive leakage-abuse attacks against symmetric searchable encryption. In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications - Volume 4: SECRYPT, (ICETE 2017)*, pages 200–211. INSTICC, 2017.

[19] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov. Breaking web applications built on top of encrypted data. In *ACM CCS 2016*, pages 1353–1364, 2016.

[20] Z. Gui, K. G. Paterson, S. Patranabis, and B. Warinschi. Swissse: System-wide security for searchable symmetric encryption. Cryptology ePrint Archive, Report 2020/1328, 2020. https://eprint.iacr.org/2020/1328.

[21] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*, 2012.

[22] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.

[23] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 94–124, 2017.

[24] S. Kamara and T. Moataz. Computationally volume-hiding structured encryption. In *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 183–213, 2019.

[25] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM CCS 2012*, pages 965–976, 2012.

[26] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim. Forward secure dynamic searchable symmetric encryption with efficient updates. In *ACM CCS 2017*, pages 1449–1463, 2017.

[27] K. Kurosawa, K. Sasaki, K. Ohta, and K. Yoneyama. UC-secure dynamic searchable symmetric encryption scheme. In *IWSEC 16*, volume 9836 of *LNCS*, pages 73–90, 2016.

[28] R. F. Lai and S. Chow. Forward-secure searchable encryption on labeled bipartite graphs. In *ACNS*, 2017.

[29] R. Lowe, N. Pow, I. Serban, and J. Pineau. The Ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. In *Proceedings of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 285–294, 2015.

[30] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. 2008.

[31] I. Miers and P. Mohassel. IO-DSSE: Scaling dynamic searchable encryption to millions of indexes by improving locality. In *NDSS 2017*, 2017.

[32] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'06, page 348–355, 2006.

[33] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *ACM CCS 2015*, pages 644–655, 2015.

[34] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic searchable encryption via blind storage. In *2014 IEEE Symposium on Security and Privacy*, pages 639–654, 2014.

[35] S. Patel, G. Persiano, K. Yeo, and M. Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *ACM CCS 2019*, pages 79–93, 2019.

[36] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 372–389, 2011.

[37] M. F. Porter. An algorithm for suffix stripping. In *Readings in Information Retrieval*. 1997.

[38] D. Pouliot and C. V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *ACM CCS 2016*, pages 1341–1352, 2016.

[39] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *In Proceedings of SIGIR'94*, pages 232–241, 1994.

[40] 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, 2000.

[41] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *NDSS 2014*, 2014.

[42] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS 2013*, pages 299–310, 2013.

[43] C. Van Rompay, R. Molva, and M. Önen. A leakage-abuse attack against multi-user searchable encryption. *PoPETs*, 2017(3):168, 2017.

[44] F. Wang, C. Tan, A. C. König, and P. Li. Efficient document clustering via online nonnegative matrix factorizations. In *Eleventh SIAM International Conference on Data Mining*, 2011.

[45] C. V. Wright and D. Pouliot. Early detection and analysis of leakage abuse vulnerabilities. Cryptology ePrint Archive, Report 2017/1052, 2017. http://eprint.iacr.org/2017/1052.

[46] A. A. Yavuz and J. Guajardo. Dynamic searchable symmetric encryption with minimal leakage and efficient updates on commodity hardware. In *SAC 2015*, volume 9566 of *LNCS*, pages 241–259, 2016.

[47] J. Zhang and T. Suel. Optimized inverted list assignment in distributed search engine architectures. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

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

[49] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6–es, 2006.

App	Type	Rank	Preview	Top-k	Update
Dropbox	$\wedge$	rel	n,d,s,p	10	✓
Box	$\wedge,\vee$	rel	n,d,s,p	6	✓
Google Drive	$\wedge$	rel	n,d,s	8	✓
Microsoft OneDrive	$\wedge$	rel	n,d,s	8	✓
Amazon Drive	$\wedge$	date	n,d,s	8	✗

Figure 12: Search features in popular storage services. Services support either just conjunctions ($\wedge$) or additionally disjunctions ($\vee$) over keywords. The top-k results are ranked according to relevance (rel) or just date. Previews may of search results may include name (n), modification date (d), file size (s), and/or the parent directory (p). Search indices may be updated due to edits within a file (✓) or only when documents are added or deleted (✗).

A Survey of Search Services

We summarize our findings in Figure 12. We do not include in the table encrypted services like Tresorit, Mega, Sync.com, and SpiderOakOne which provide client-side encryption but only currently allow search of (unencrypted) filenames (for the first three) or no search at all (for SpiderOakOne).

B Lucene Details and Configuration

We provide details on Lucene that are relevant to our experiments in Section 3, and, for other detailed aspects, please refer to its documentation [1].

Lucene breaks the index into multiple sub-indices called *segment*s, each of which is a standalone index. Incoming updates are always added to the current opening segment in memory, and the opening one is committed to disk when closed or reaching the threshold on size. Depending on the workloads, the segments can be merged offline, e.g., when the number of segments reaches some threshold; or simply after every update. Lucene provides two ranking options, a TF-IDF relevance function and a BM25 relevance function with default $k_1 = 1.2$ and $b = 0.75$.

Lucene provides multiple index encoding implementations, or *codec*s. For Lucene 7.7.3, the default one is `Lucene50`, which encodes the inverted index in separate files for term index, term dictionary and postings, and applies the delta encoding and variable-byte encoding on numbers, e.g., identifier and term frequency, in the index. The `Lucene50` codec also implements the *skip list* technique to enable fast posting access in a posting list. In addition, `Lucene50` applies `LZ4` compression on the forward index, but not on the inverted index. The simplest codec available in Lucene is `SimpleTextCodec`, which is a text-based index encoder that serializes the terms and postings into one big string, without any optimization.

For our experiments in Section 3 we configured Lucene to immediately merge segments after each update and also disabled the skip list feature. We see no reason why these features would prevent attacks, but they appear to make them somewhat more difficult. We also configure Lucene to not include the document positions in the postings since this information is irrelevant to our target queries.