Dynamic Searchable Encryption via Blind Storage

Muhammad Naveed, Manoj Prabhakaran, Carl A. Gunter
University of Illinois at Urbana-Champaign

***Abstract*—Dynamic Searchable Symmetric Encryption allows a client to store a dynamic collection of encrypted documents with a server, and later quickly carry out keyword searches on these encrypted documents, while revealing minimal information to the server. In this paper we present a new dynamic SSE scheme that is simpler and more efficient than existing schemes while revealing less information to the server than prior schemes, achieving fully adaptive security against honest-but-curious servers.**

We implemented a prototype of our scheme and demonstrated its efficiency on datasets from prior work. Apart from its concrete efficiency, our scheme is also simpler: in particular, it does not require the server to support any operation other than upload and download of data. Thus the server in our scheme can be based solely on a cloud storage service, rather than a cloud computation service as well, as in prior work.

In building our dynamic SSE scheme, we introduce a new primitive called *Blind Storage*, which allows a client to store a set of files on a remote server in such a way that the server does not learn how many files are stored, or the lengths of the individual files; as each file is retrieved, the server learns about its existence (and can notice the same file being downloaded subsequently), but the file's name and contents are not revealed. This is a primitive with several applications other than SSE, and is of independent interest.

I. Introduction

In recent years, searchable symmetric encryption (SSE) has emerged as an important problem at the intersection of cryptography, cloud storage, and cloud computing. SSE allows a client to store a large collection of encrypted documents with a server, and later quickly carry out keyword searches on these encrypted documents. The server is required to not learn any more information from this interaction, beyond certain patterns (if two searches involve the same keyword, and if the same document appears in the result of multiple searches, but not the actual keywords or the contents of the documents).

A long line of recent work has investigated SSE with improved security, more flexible functionality and better efficiency [23], [10], [18], [17], [6]. The techniques in all these works build on the early work of [10], [9]. In this work we present a radically different approach that achieves stronger security guarantees and flexibility, with significant performance improvements. In particular, our construction enjoys the following features:

- Dynamic SSE, which supports adding and removing documents at any point during the life-time of the system.
- The server is "computation free". Indeed, the only operations that need to be supported by the server are uploading and downloading blocks of data, if possible, parallelly. This makes our system highly scalable, and any optimizations in these operations (e.g., using a content delivery network) will be directly reflected in the performance of the system.
- The information revealed to the server ("leakage functions") is strictly lesser than in all prior Dynamic SSE schemes except [24]. Scheme of [24] reveals less information to the server at the expense of poly-logarithmic overhead on top of Dynamic SSE overhead of other schemes (including ours).
- Satisfies a fully adaptive security definition, allowing for the possibility that the search queries can be adversarially influenced based on the information revealed to the server by prior searches.
- Security is in the standard model, rather than the heuristic Random Oracle Model; relies only on the security of block ciphers and collision resistant hash functions.
- Optional *document-set privacy*. The number of documents in the system and their lengths can be kept secret, revealing the existence of a document only when it is accessed by the client (typically after learning that a keyword appears in that document). This allows one, for instance, to archive e-mail with support for keyword searching, while keeping the number and lengths of e-mails hidden from the server (until each one is retrieved).

A simple prototype has been implemented to demonstrate the efficiency of the system.

Blind Storage. An important contribution of this work is to identify a more basic primitive that we call *Blind Storage*, on which our Dynamic SSE scheme is based. A Blind Storage scheme allows a client to store a set of files on a remote server in such a way that the server does not learn how many files are stored, or the lengths of the individual files; as each file is retrieved, the server learns about its existence (and can notice the same file being downloaded subsequently), but the file's name and contents are not revealed. Our Blind Storage scheme also supports adding new files and updating or deleting existing files. Further, though not needed for the Dynamic SSE construction, our Blind Storage scheme can be used so that the actual operation — whether it is reading, writing, deleting or updating — is hidden from the server.

Though not the focus of this work, we remark that a Blind Storage system would have direct applications in itself, rather than as a tool in constructing flexible and efficient Dynamic SSE schemes. As our Blind Storage scheme does not make requirements on the server other than storage, it can be used with commodity storage systems such as Dropbox. This enables a wide range of simple applications that can take advantage of modular privacy protections to operate at a large scale and low expense but with strong privacy guarantees. Applications can range from backing up a laptop to archiving patient records at a hospital. Further, in our dynamic SSE scheme, document set privacy with relatively low overhead is made possible because we can simply store all the documents in the same Blind Storage system that is used to implement the SSE scheme.

II. Related Work

The problem of searching on encrypted data has received increasing attention from the security and cryptography community, with the growing importance of cloud storage and cloud computation. One of the major hurdles in outsourcing data storage and management for businesses has been security and privacy concerns [15], [21], [3]. Theoretical cryptography literature offers an extremely powerful and highly secure solution in the form of Oblivious Random Access Memory (ORAM) [20], [13], which addresses almost all of the security concerns related to storing data in an untrusted server. However, this solution remains very inefficient for several important applications, despite significant recent improvements [22], [26], [25]. The notion of Symmetric Searchable Encryption (SSE) — investigated in a long line of works including [23], [11], [7], [10], [27], [8], [19], [18], [17], [24], among others — attempts to strike a different balance between efficiency and security, by letting the server learn just the *pattern* of data access (and ideally, nothing more), in return for a simpler and faster construction; further, one often settles for security against passively corrupt (honest-but-curious) servers. The scheme of [24] also provides a notion of forward privacy, which prevents leaking whether a newly added document contains the keywords the user has already searched for.

The approach in [10] formed the basis for many subsequent works. The basic idea is to use an index that maps each search keyword to the list of documents that contains it. This list is kept as an encrypted linked list, with each node containing the key to decrypt next node. The nodes of all the linked lists are kept together, randomly sorted. Until the head of a linked list is decrypted, it is virtually invisible to the server; in particular, the number of linked lists and their lengths remain hidden from the server. This construction provided non-adaptive security (which assumes that all the search queries are generated at once); efficiently achieving adaptive security has been the subject of much research starting with [10].

An important aspect of SSE is whether it is *dynamic* or not: i.e., whether the client can update the document collection after starting to search on it. Dynamic SSE schemes were presented in [11], [27], [18], [17], [5], [24].

Finally, we mention a few variants of the SSE problem that are *not* considered in this work. One could require security against actively corrupt servers, rather than just honest-but-curious servers. Another variant requires more expressive searches, involving multiple keywords (e.g., [14], [6], [5]). One could also require that many clients can perform searches on a document collection created by a single data-owner [16]. While we do not consider these problems in this work, the main new tool we build — namely, a Blind Storage system — is a general-purpose tool and is likely to be useful for expressive search queries. Indeed, it could be used to implement components like the "T-sets" of [6] more efficiently. These and other extensions are subject of on going work.

III. Technical Overview

In this section, we briefly discuss our techniques and the advantages of our scheme compared to prior SSE constructions. Most of the advantages follow from the simplicity of our scheme, and in particular, as depicted in Figure 1, from the fact that our server is computation free.

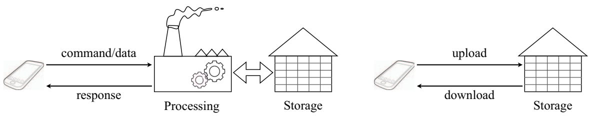

Fig. 1: Contrasting the architecture of existing SSE Schemes (on the left) with that of the proposed scheme.

Techniques. Our main construction is that of a versatile tool called Blind Storage, which is then used to build a full-fledged SSE scheme. A Blind Storage scheme lets the client keep all information — including the number and size — about files secret from the server storing them, until they are accessed. In building the SSE scheme, the *search index* entries for all the keywords are stored as individual files in the Blind Storage scheme (with care taken to facilitate updates).

Our Blind Storage scheme, called ScatterStore, is constructed using a simple, yet powerful technique: each file is stored as a collection of blocks that are kept in pseudorandom locations; the server sees only a *super-set* of the locations where the file's blocks are kept, and not the exact set of locations.[1] The key security property this yields us is that, from the point of view of the server, each file is associated with a set of locations *independent of the other files* in the system. (Indeed, the sets of locations for two files can overlap.)

A rigorous probabilistic analysis shows that for appropriate choice of parameters, the probability that any information about files not yet accessed is leaked to the server can be made negligible (say, 2^{-40} or 2^{-80}), with a modest blow-up in the storage and communication costs (e.g., by a factor of 4) over unprotected storage.

The only cryptographic tools used in our scheme are block ciphers (used for standard symmetric key encryption as well as for generating pseudorandom locations where the data blocks are kept) and collision resistant hash functions. The security parameters for these tools are chosen independently of the other parameters in the scheme.

Architecture. Most of the previous SSE schemes were presented as using a dedicated server, that performed both storage and computation. (See Figure 1.) The computation typically involved an (unparallelizable) sequence of decryptions. To deploy such a scheme using commodity services, one would need to rely not only on cloud storage services, but also cloud computation services. This presents several limitations. Firstly, this limits the choice of service providers available to a user: one could use Amazon EC2 for computation, combined with Amazon S3 for storage; however, it is not viable to use Dropbox for persistent storage and Amazon EC2 for computation, as this would incur high costs for communication between these two services. Storage and compute clusters are physically separated in modern data centers. This would add additional latency in all dynamic SSE schemes except ours, as data needs to be transmitted from storage nodes to compute nodes over the data center network. In contrast, our system

[1] To the extent that extra blocks are read, our scheme is similar to existing Oblivious RAM constructions. However, in our case, the overhead of extra blocks is bounded by a constant factor.

can be easily implemented using Dropbox or other similar services which provide only storage. Secondly, relying on cloud computation makes the deployment less flexible, as it is harder to change choices like that of the operating system (due to pricing changes or technical support, for instance).

Another important issue in existing schemes is that one relies on availability and trust assumptions (e.g., honest-but-curious) for both computation and storage. Clearly, it is desirable to trust storage alone, as is the case in our scheme. Further, in ongoing work, we consider obtaining security against actively corrupt (rather than honest-but-curious) servers; this is easier and more efficient to achieve starting from our scheme, since we need to enforce honest behavior on part of a server that provides storage alone.

Finally, it is significantly cheaper to rely on a cloud-storage service alone than on cloud computation (plus persistent storage).

Security definition. An important feature of our schemes is the stronger and easier to understand security guarantees. All the information leaked to the server is fully captured in relatively simple functionalities. For the Blind Storage scheme, as shown in Figure 2, each time a file is accessed, the functionality $\mathcal{F}_{\text{STORE}}$ reveals just a triple $(\mathsf{op}, j, \mathsf{size})$ to the server, where op specifies what the access operation is (read, write, update or delete), j specifies the last time, if any, the same file was accessed, and size specifies the size of the file.

The functionality $\mathcal{F}_{\text{SSE}}$ (shown in Figure 7) specifies all the information revealed by our SSE scheme. It is slightly more complex, partly because it allows the client to reuse document IDs. Further, it offers a higher level of secrecy for documents that are originally in the system, compared to those added later during the operation of the system.

Fully Adaptive Security. As mentioned in Section I, we achieve fully adaptive security, without relying on heuristics like the random oracle model. Technically, this is a consequence of the fact that the server does not carry out any decryptions. We point out that achieving adaptive security by making the client do decryptions for the server would not be viable in existing SSE schemes because a long sequence of decryptions (that cannot be parallelized) need to be carried out; several rounds of communication (with attendant network delays) would be necessary if the client carries out these decryptions for the server. Nevertheless, a similar approach was mentioned in [6] as a theoretical solution to avoid the Random Oracle Model and retain adaptive security.

The price we pay for the improved security, greater computational efficiency, parallelizability and simpler architecture is that the server storage and communication costs are possibly higher than that of some of the existing schemes (e.g., a factor of 2 to 4 over unprotected storage, which is in fact, comparable to overheads incurred in some other schemes like that of [6]). Also our SSE scheme could, in principle, involve up to three rounds of communication for retrieving the documents (this happens if the keyword has a large number of matching documents). In contrast, many existing schemes involve only two rounds (one to retrieve encrypted list of documents, and one to retrieve the documents themselves).

Comparative Performance. The most natural prior work for us to compare against is [18] (though, unlike this work, it uses the Random Oracle Model). We remark that the more recent work of [6] augments the functionality of [18] (but without support for dynamic updates), and provides a highly streamlined implementation over very large scale data; however, *for the task of simple keyword searches,* its algorithm remains comparable to [18]. Since [18] reports performance of a prototype implemented in a comparable environment as ours (conservative comparison: we use a laptop and they used a server), we compare with it. Asymptotically, the client-side storage and computation in our system is same as [18], but the constants for our scheme are much better, and is reflected in the performance measured. Our scheme completely avoids server-side computation (which is quite significant in [18]).

IV. Blind Storage

As mentioned in Section I, an important contribution of this paper is to identify a versatile primitive called Blind Storage. It allows a client to store a set of files with a remote server, revealing to the server neither the number nor the sizes of the files. The server would learn about the existence of a file (and its size, but not the name used by the client to refer to the file, or its contents) only when the client retrieves it later. We also allow the client to add new files, and to update or delete existing files. The client's local storage should be independent of the total amount of data stored in the system.

In this section, first we present the definition of a Blind Storage system, followed by an efficient construction SCATTERSTORE, and a proof of security. Later, in Section V-B, we show how to build a Dynamic SSE scheme using a Blind Storage system.

A. Definition

Below, first we define the syntax of a Blind Storage system (and the infrastructure it needs), followed by the security requirements on it.

The Syntax. A blind storage system consists of a client and a "dumb" storage server. The server is expected to provide only two operations, $\mathsf{download}$ and upload. The data is represented as an array of *blocks*; the $\mathsf{download}$ operation is allowed to specify a list of indices of blocks to be downloaded; similarly, the upload operation is allowed to specify a list of data blocks and indices for those blocks.

A blind storage system is defined by three polynomial-time algorithms on the client-side: BSTORE.Keygen, BSTORE.Build and BSTORE.Access. Of these, BSTORE.Access is an interactive protocol.

- BSTORE.Keygen takes security parameter as an input and outputs a key $\mathsf{K}_{\text{BSTORE}}$ (typically a collection of keys for the various cryptographic primitives used). Note that $\mathsf{K}_{\text{BSTORE}}$, which the client is required to retain throughout the lifetime of the system, is required to be independent of the data to be stored.
- BSTORE.Build takes as input $(\mathsf{K}_{\text{BSTORE}}, d_0, \{\mathsf{id}_i, \mathsf{data}_i\}_{i=1}^{t}$, where $\mathsf{K}_{\text{BSTORE}}$ is a key, d_0 is an upperbound on the total number of data blocks to be stored in the system, $(\mathsf{id}_i, \mathsf{data}_i)$ are the id and data of the files that the system to be initialized

- On receiving the command $\mathcal{F}_{\textsc{store}}.\mathsf{Build}$ from the client:
    - ○ $\mathcal{F}_{\textsc{store}}$ accepts input $(d_0, \{\mathsf{id}_i, \mathsf{data}_i\}_{i=1}^{t})$ from the client (where d_0 is an upperbound on the total number of data blocks to be stored in the system at any time, and the rest specify files to be stored in the system initially); it internally stores the specified files.
    - ○ **Build Leakage:** In addition, $\mathcal{F}_{\textsc{store}}$ sends d_0 to the server.
- On receiving the command $\mathcal{F}_{\textsc{store}}.\mathsf{Access}(\mathsf{id}, \mathsf{op})$ from the client:
    - ○ If no file matching the identifier id exists, and the operation $\mathsf{op} \in \{\mathsf{read}, \mathsf{delete}\}$, $\mathcal{F}_{\textsc{store}}$ returns a status message to the client indicating so. Else, if $\mathsf{op} = \mathsf{read}$, $\mathcal{F}_{\textsc{store}}$ returns the file with identifier id; if $\mathsf{op} = \mathsf{delete}$, it is removed. If $\mathsf{op} = \mathsf{write}$, the content data for the file is also accepted from the client, and the file is created or its content replaced with data. If $\mathsf{op} = \mathsf{update}$, $\mathcal{F}_{\textsc{store}}$ interacts with the client as follows:
        - ▪ $\mathcal{F}_{\textsc{store}}$ returns the current size of the file (in blocks – possibly 0, if the file does not exist) to the client.
        - ▪ $\mathcal{F}_{\textsc{store}}$ accepts the size of the updated file from the client.
        - ▪ $\mathcal{F}_{\textsc{store}}$ returns the current contents of the file to the client.
        - ▪ $\mathcal{F}_{\textsc{store}}$ accepts the updated contents of the file from the client. The file stored internally is updated with this.
    - ○ **Access Leakage:** In addition, $\mathcal{F}_{\textsc{store}}$ sends the tuple $(\mathsf{op}, j, \mathsf{size})$ to the server where:
        - ▪ op specifies what the current access operation is,[a]
        - ▪ j is the last instance when the same file was accessed ($j = 0$ means that this file was not accessed before)
        - ▪ size is the size (in number of blocks) of the file being accessed. For the update operation, size is the larger of the sizes before and after the update.

[a] A refined version of Blind-Storage would require the operation to be not revealed. See Section IV-B3.

Fig. 2: The $\mathcal{F}_{\textsc{store}}$ functionality: all the information leaked to the server in our Blind Storage scheme is specified here.

with; it outputs an array of blocks D to be uploaded to the server.

- BSTORE.Access takes as input a key $\mathsf{K}_{\textsc{bstore}}$, a file id id, an operation specifier $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}, \mathsf{update}, \mathsf{delete}\}$, and optionally data data (if op is write or update). Then it interacts with the server (through the upload/download interface) and returns a status message and optionally file data (for the read and update operations). For the update operation, BSTORE.Access allows more flexibility:[2] first it requires only id as input, and outputs the current size of the file with that ID; then it accepts as input (an upperbound on) what the size of the file will be after update; then it outputs the current file data, and only then requires the new data with which the file will be updated.

Security Requirement. We specify the security requirement of a blind-storage system following the "real/ideal" paradigm that is standard for secure multi-party computation (as opposed to using specific game-based security definitions used in some of the earlier literature on SSE). This includes specifying an adversary model and an "ideal functionality," as detailed below. The formal security requirement we shall require is that of Universally Composable security [4] (but restricted to our adversary model).[3]

In the adversary model we consider, the adversary is allowed to corrupt only the server *passively* — i.e., as an honest-but-curious adversary. (If the client is corrupt, we need not provide any security guarantees.)

The ideal functionality is specified as a virtual trusted third party $\mathcal{F}_{\textsc{store}}$ that mediates between the client and the server (modeling the information leaked to the server). $\mathcal{F}_{\textsc{store}}$ accepts two commands from the client: $\mathcal{F}_{\textsc{store}}.\mathsf{Build}$ and $\mathcal{F}_{\textsc{store}}.\mathsf{Access}$, along with inputs to these commands (which are identical to the inputs to BSTORE.Build and BSTORE.Access as described above, except for the key $\mathsf{K}_{\textsc{bstore}}$). In this ideal model, it is $\mathcal{F}_{\textsc{store}}$ which maintains the collection of files, and performs all the operations specified by the $\mathcal{F}_{\textsc{store}}.\mathsf{Build}$ and $\mathcal{F}_{\textsc{store}}.\mathsf{Access}$ commands. In addition, it reveals limited information to the server as specified in Figure 2.

We stress that *all the information revealed to the server by our blind-storage scheme is captured by the $\mathcal{F}_{\textsc{store}}$ functionality.* Note that the information leaked (during $\mathcal{F}_{\textsc{store}}.\mathsf{Build}$ and $\mathcal{F}_{\textsc{store}}.\mathsf{Access}$) is limited and simple to specify. This simplicity is one of the important contributions of this work.

Remark. Even when using the ideal $\mathcal{F}_{\textsc{store}}$ functionality, an adversary can learn some statistics about the files and accesses by analyzing the patterns in the information revealed to it. Such information could indeed be sensitive, and it is up to the higher-level application that uses a blind-storage system to ensure that this is not the case. The cryptographic construction seeks to only match the guarantees given by $\mathcal{F}_{\textsc{store}}$.

B. Our Construction

Our Blind Storage construction is called SCATTERSTORE. First, we shall present a simplified version, called SCATTERSTORE-LITE, which already involves most of the critical components in the full construction. The only drawback of the simplified construction is that the client is required to maintain a data structure to map each file-name to a small piece of information. This solution is well-suited for a scenario when the system consists of a moderate number of large files. In our final construction, we show how to avoid this local data structure, so that the client's storage is of constant size, independent of the number of files in the system.

1) Simplified Construction: SCATTERSTORE-LITE*:* In this section, we present a sketch of SCATTERSTORE-LITE, our simplified Blind-Storage construction. We defer a formal description to the next section where we present the full construction.

[2] One can always use a read followed by a write to get the effect of an update, but this is less efficient and potentially reveals more information.

[3] We remark that for our setting of passive adversaries, UC security is a conceptually simpler notion than for the setting of active adversaries. Nevertheless, for the sake of concreteness, we use the UC security model, which automatically ensures security even when the inputs to the client are adaptively chosen under adversarial influence.

The construction relies on the following primitives:

- a full-domain collision resistant hash function (CRHF), H,
- a pseudorandom function (PRF), Φ,
- a *full-domain* pseudorandom function (FD-PRF), Ψ (implemented by applying Φ to the output of H),
- a pseudorandom generator (PRG), Γ.

(In our prototype, as described in Section VI, the implementation of Φ, Ψ and Γ all rely on the AES block-cipher; H is implemented using SHA-256.) The security parameter k is an implicit input to all the cryptographic primitives used in the construction. The other parameters in the construction are the size parameters n_{D}, m_{D}, an expansion parameter $\alpha > 1$, and the minimum number of blocks communicated in each transaction, κ.

- BSTORE.Keygen: A key K_Φ for the PRF Φ, and a key K_{ID} for the FD-PRF Ψ are generated; $\mathsf{K}_{\text{BSTORE}}$ is set to be the pair $(\mathsf{K}_\Phi, \mathsf{K}_{\mathrm{ID}})$.
- BSTORE.Build$(\mathbf{F}, \mathsf{K}_{\text{BSTORE}})$: $\mathbf{F}$ is a list of files $\mathsf{f} = (\mathsf{id}_\mathsf{f}, \mathsf{data}_\mathsf{f})$. Below size_f denotes the number of blocks in an encoding of data_f; each block has two short header fields containing a *version number* initialized to 0, and $H(\mathsf{id}_\mathsf{f})$; the latter is not allowed to be all 0s, which is reserved to indicate a free block. In addition, the first block has a header field that records size_f. (It will be convenient to keep the version number field at an extreme end of the block, as it needs to kept unencrypted, whereas the rest of the block will be encrypted at the end of this phase.)
  - ○ Let D be an array of n_{D} blocks of m_{D} bits each.
  - ○ Initialize every block in D with all 0s (to be encrypted later).
  - ○ For each file f in $\mathbf{F}$,
    1) Generate a pseudorandom subset $S_\mathsf{f} \subseteq [n_\mathsf{D}]$, of size $|S_\mathsf{f}| = \max(\lceil \alpha \cdot \mathsf{size}_\mathsf{f} \rceil, \kappa)$ as follows.
       a) Generate a seed $\sigma_\mathsf{f} = \Psi_{\mathsf{K}_{\mathrm{ID}}}(\mathsf{id}_\mathsf{f})$ for the PRG Γ.
       b) Let S_f be the set of integers in the sequence $\Lambda[\sigma_\mathsf{f}, |S_\mathsf{f}|]$. Here $\Lambda[\sigma, \ell]$ denotes a sequence of ℓ integers obtained as follows.
          - Generate a (sufficiently long) output from the PRG Γ, with seed σ, and parse it as a sequence of integers in the range $[n_\mathsf{D}]$.
          - $\Lambda[\sigma, \ell]$ is the first ℓ distinct integers in this sequence.
    2) Check if the following two conditions hold:
       - at least size_f blocks in D that are indexed by the numbers in S_f are free;
       - at least one block in D that is indexed by the numbers in S_f^0 is free.

       If either condition does not hold, *abort*. By the choice of our parameters, this will happen only with negligible probability.
    3) Pick a pseudorandom subset $\widehat{S}_\mathsf{f} \subseteq S_\mathsf{f}$ of size $|\widehat{S}_\mathsf{f}| = \mathsf{size}_\mathsf{f}$, such that the blocks in D that are indexed by the numbers in $\widehat{S}_\mathsf{f}$ are all free. For convenience, we shall rely on the fact that the numbers in the sequence used to generate S_f are in a pseudorandom order; we pick the shortest prefix of this sequence that contains size_f numbers indexing free blocks, and let $\widehat{S}_\mathsf{f}$ be the set of these size_f numbers.
    4) Write the size_f blocks of data_f onto the blocks in D that are indexed by the numbers in $\widehat{S}_\mathsf{f}$ (in increasing order). These blocks get marked as not free.
  - ○ Encrypt each block of D using the PRF Φ and the key K_Φ. The version number field is left unencrypted, while the rest is encrypted using the version number (initialized to 0) and the index number of the block as IV. More precisely, for the i^{th} block $\mathsf{D}[i]$, we split it as $v_i||\mathsf{B}[i]$ (v_i being the version number), and then update $\mathsf{B}[i]$ to $\mathsf{B}[i] \oplus \Phi_{\mathsf{K}_\Phi}(v_i||i)$.
    (If the block-size of the PRF is less than the size of the block $\mathsf{B}[i]$, then a few lower-order bits of the IV are reserved for use as a counter, to obtain multiple blocks from the PRF for a single block in D.)

Fig. 3: SCATTERSTORE: A Blind-Storage Scheme (continued on next page)

In our construction, each file in the blind storage system is kept in a large array D of encrypted blocks, at positions indexed by a *pseudorandom set*. This set is defined by a short seed and the size of the set: the seed can be used to generate a (virtually infinite) pseudorandom sequence, and the size specifies the length of the prefix of this sequence that defines S_f. In our simplified construction, the client stores this information in a data-structure that maps the file-names to the descriptor of the pseudorandom set.[4]

The main security property that we need to ensure is that the location of the blocks of one file does not reveal any information about the blocks of the other files, or even the proportion of occupied and free blocks in D.[5] However, clearly, we cannot choose the positions to store blocks of one file independent of the blocks of the other files, since two files must not occupy the same block. A naïve solution would be to use a large D, to reduce the probability that the blocks chosen for one file do not overlap with that for any other file. But this is problematic, because to reduce the probability of such a collision to a small quantity (say, negligible in the security parameter), size of D needs to be enormously larger (i.e., a super-polynomial factor larger) than the actual amount of data stored.

We overcome this inherent tension between collision probability and wasted space as follows. To store a file f of n blocks, we choose a pseudorandom subset S_f of not n blocks, but say (for a typical setting of parameters), $2n$ blocks. This subset of $2n$ blocks will be chosen independent of the other files in the system (and it is this subset that the server sees when the client accesses this file). Within this set we choose a subset $\widehat{S}_\mathsf{f} \subseteq S_\mathsf{f}$ of n blocks, where the actual data is stored. The set $\widehat{S}_\mathsf{f}$ is of course, selected depending on the other blocks used by other files, to avoid collisions. However, since the contents of the blocks are kept encrypted, the server does not learn anything about $\widehat{S}_\mathsf{f}$ (except its size).

This, it turns out, allows D to be only a small constant factor larger than the total data to be stored in the system.

[4] Only the *size* of the pseudorandom set needs to be stored. The seed for the set can be derived by applying a (full-domain) pseudorandom function to the file-name. See next section.

[5] This property manifests itself in the simulation based proof of security, since the simulator will pick the locations of blocks of a file being accessed independent of the number and size of files that are not yet accessed.

- BSTORE.$\mathsf{Access}(\mathsf{id}_\mathsf{f}, \mathsf{op}, \mathsf{K}_{\text{BSTORE}})$: We describe the case when $\mathsf{op} = \mathsf{update}$, and mention how the other operations differ from it.
  1) First, compute $\sigma_\mathsf{f} = \Psi_{\mathsf{K}_{\mathrm{ID}}}(\mathsf{id}_\mathsf{f})$, and define the set S_f^0 of size κ, to consist of the numbers in the sequence $\Lambda[\sigma_\mathsf{f}, \kappa]$. Retrieve the blocks indexed by S_f^0 from D.
  2) Decrypt the blocks of $\mathsf{D}[S_\mathsf{f}^0]$ (where $\mathsf{D}[i] = (v_i||\mathsf{B}[i])$ is decrypted as $\mathsf{B}[i] \oplus \Phi_{\mathsf{K}_\Phi}(v_i||i)$), in the order in which they appear in $\Lambda[\sigma_\mathsf{f}, \kappa]$, until a block which is marked as belonging to id_f is encountered. If no such block is encountered the file is not present in the system. In this case, set $\mathsf{size}_\mathsf{f} = 0$.
  3) Otherwise (if a block marked as belonging to id_f is found), this is the first block of the file with identifier id_f: recover the size of the file size_f from the header of this block.
  4) Output size_f to the client and accept as input $\mathsf{size}'_\mathsf{f}$, the size of the file after update.
  5) Let $\ell = \lceil \alpha \cdot \max(\mathsf{size}_\mathsf{f}, \mathsf{size}'_\mathsf{f}) \rceil$. If $\ell \leq \kappa$, let $S_\mathsf{f} = S_\mathsf{f}^0$. Else, let S_f be the set of numbers in $\Lambda[\sigma_\mathsf{f}, \ell]$. In this case, retrieve the blocks indexed by $S_\mathsf{f} \setminus S_\mathsf{f}^0$ from the server.
  6) Some of the blocks indexed by S_f would have already been decrypted in Step 2 above. Decrypt the remaining blocks indexed by S_f, as well.
  7) Identify $\widehat{S}_\mathsf{f}$ as the set of indices of blocks belonging to the file being accessed (by checking if their headers match id_f). If $\widehat{S}_\mathsf{f}$ is not empty, combine these blocks together (in increasing order of their indices) to recover the entire contents of the file, and output it.
  8) Accept as input new contents data' encoded as $\mathsf{size}'_\mathsf{f}$ blocks.
  9) Identify a subset $\widehat{S}'_\mathsf{f} \subseteq S_\mathsf{f}$ of size $\mathsf{size}'_\mathsf{f}$ as follows. Find the shortest prefix of the sequence $\Lambda[\sigma_\mathsf{f}, \ell]$ which contains $\mathsf{size}'_\mathsf{f}$ blocks that are either marked as belonging to id_f (i.e., in $\widehat{S}_\mathsf{f}$) or are free.
     If no such prefix exists, or if the first of the $\mathsf{size}'_\mathsf{f}$ blocks identified is not within $\Lambda[\sigma_\mathsf{f}, \kappa]$ (this can happen only when $\mathsf{size}_\mathsf{f} = 0$), then *abort*; again, by the choice of our parameters, this will happen only with negligible probability.
     Note that if $\mathsf{size}'_\mathsf{f} < \mathsf{size}_\mathsf{f}$, then $\widehat{S}'_\mathsf{f} \subseteq \widehat{S}_\mathsf{f}$; else, $\widehat{S}_\mathsf{f} \subseteq \widehat{S}'_\mathsf{f} \subseteq S_\mathsf{f}$.
  10) Then update the blocks indexed by $\widehat{S}'_\mathsf{f}$ with the blocks of data'. If $\mathsf{size}'_\mathsf{f} < \mathsf{size}_\mathsf{f}$ mark as free the blocks indexed by $\widehat{S}_\mathsf{f} \setminus \widehat{S}'_\mathsf{f}$.
  11) Encrypt all the blocks indexed by S_f using the IV $v_i||i$ as described in the BSTORE.Build step, but after incrementing v_i for each block.
  12) Upload the newly reencrypted blocks back to the server. Note that all the blocks that were downloaded, i.e., $\mathsf{D}[S_\mathsf{f}]$, are uploaded back, with their version numbers incremented by 1, and reencrypted.
- When $\mathsf{op} = \mathsf{read}$, the Steps 1 through 7 from above are carried out, but setting $\mathsf{size}'_\mathsf{f} = 0$.
- When $\mathsf{op} = \mathsf{write}$, the behavior is the same as when $\mathsf{op} = \mathsf{update}$, except that the new file data is taken as input upfront, and no data is returned.
- When $\mathsf{op} = \mathsf{delete}$, the behavior is the same as when $\mathsf{op} = \mathsf{write}$, except that it takes $\mathsf{size}'_\mathsf{f} = 0$.

Fig. 4: SCATTERSTORE: A Blind-Storage Scheme (continued from Figure 3)

A typical parameter setting would be to let D have 4 times as many blocks as total data blocks to be stored. Then, we can drive the information leaked to the server to a negligible quantity with only small constant factor overheads in the storage and communication.[6]

An important feature of our pseudorandom set construction, compared to linked-list based construction of related work in the literature, is that the server need not carry out any decryptions. In linked-list based constructions, each node in the list is progressively revealed; even if the server were to take the help of the client in decrypting each node, several rounds of communication will be required.[7] In contrast, our construction allows the server to be "crypto-free" and still have only constant number of rounds of interaction.

This simplicity leads to another advantage: our construction meets a fully adaptive security definition for blind storage (and for searchable encryption) against honest-but-curious servers. Here, adaptive security refers to the fact that the choice of which files the client needs to access can be adversarially influenced, after the system has been deployed. Prior work required less efficient and more complicated schemes to achieve adaptive security, and often employed the Random Oracle heuristics [18], [10]. We use only standard primitives (PRFs and collision-resistant hash functions) and obtain security in the standard model (i.e., not in the Random Oracle model).

Finally, our pseudorandom set construction easily supports a *dynamic* blind-storage scheme. We sketch the update operation (creating and deleting a file are essentially special cases of the update operation). To update a file, the client retrieves the encrypted blocks corresponding to the file's pseudorandom set S_f, decrypts them, updates the subset of blocks $\widehat{S}_\mathsf{f}$ where the file's blocks are present, reencrypts all of the downloaded blocks (i.e., all of S_f), and uploads them back to the server. There are two details worth highlighting:

- Encryption of each block is carried out by XOR-ing the contents of the block with the output of a PRF, which keyed using a fixed secret key, but whose inputs depend on the block: this input consists of the block's index in D and its current *version number*. (The version number is specific to each block, and it is kept unencrypted in the block.) Initially, all blocks have version number 0, and when reencrypting a block, its version number is incremented.
- In the above process, the updated file may need fewer or more blocks than the original file. We let the size of the set S_f that is retrieved to correspond to the longer of the two versions of the file. If the updated version needs fewer blocks than the original file (in which case S_f corresponds to the original file), the extra blocks are marked as free. If the updated file needs more blocks, then the subset S_f that is retrieved corresponds to the size of the file after the update; then, additional empty blocks are located in S_f to extend $\widehat{S}_\mathsf{f}$ to be large enough for the updated file. In either case, the

[6] We note that the pseudorandom set S_f would have to be at least a minimum size κ (say, $\kappa = 80$ blocks); when accessing small files which are just a few blocks long (i.e., n is small), $2n$ will be less than this minimum. For such files, the *communication* involves an additive overhead equal to this minimum. However, the number of blocks occupied in D, i.e., size of $\widehat{S}_\mathsf{f}$, is always n, irrespective of n being small or large.

[7] In our full construction, the server does take the help of the client to carry out a decryption and to recover the description of the pseudorandom set; but this involves only one round of communication.

server sees the size of the larger of the two versions.

2) Full Construction: SCATTERSTORE*:* We present the details of our final Blind Storage scheme in Figure 3 and Figure 4. Here we give a brief sketch of the main ideas.

In the simpler scheme above, we allowed the client to maintain a data-structure mapping a file-identifier id_f to a descriptor of the pseudorandom set S_f. This is not desirable if the system would store a large number of small files; then the size of this data structure is comparable to that of the entire collection of files. We would like our client to store only a constant number of cryptographic keys, so that its storage requirement does not grow with the size of the entire set of files stored in the system.

For this, recall that the two pieces of information needed to define a pseudorandom set S_f are a seed and the size of the set. The seed itself can be obtained by applying a full-domain PRF to the file-identifier. (A full-domain PRF can be implemented using a full-domain collision resistant function (CRHF) and a normal PRF: an arbitrary-length file-identifier is first hashed to a fixed-length input for the PRF using the CRHF.) If the client knew the size of S_f as well, there will be no need to store this map at all. We exploit this to use a two-level access to a file, as follows.

For each file, the first block consists of a header that stores the size (number of blocks) of the file. To retrieve a file id_f, the client assumes that the file is "small" and retrieves a pseudorandom set S_f^0 with the smallest possible number of blocks, i.e., κ. After recovering the first block of the file from the blocks in S_f^0, the client computes the actual size of S_f and if it is larger than κ, then retrieves the rest of S_f from the server. (Note that S_f is simply a superset of S_f^0, obtained from a longer pseudorandom sequence.) We remark that it is important for security that when $|S_\mathsf{f}| > \kappa$ the client performs this second access, even if the entire file happened to fit within the blocks in S_f^0.

The update functionality as we have defined, fits well into this two-level access. To update a file id_f, first the client is allowed to learn the current size of the file before providing any information about the update; this size information is retrieved after the first level of access and returned to the client. (Note that we could have in fact provided the client with the first few blocks of the current file too, but for simplicity we omit this from the specification of the functionality.) Next, before the second level of access, the larger of the current file size and updated file size needs to be known. So at this point, we require the client to submit the size of the updated file. Then the size of the set S_f to be retrieved is defined by the larger of the current and updated sizes. If this set has more than κ blocks, the second level of access retrieves the remaining blocks; then, as in the simpler construction, all the retrieved blocks will be reencrypted (with a subset of them having updated contents) and uploaded back on the server.

3) Variations and Enhancements: There are several optimizations and variations to this construction that would be of interest. We mention a few.

- The time taken for the read operation can be significantly improved as follows. As presented above, in reading file, the client retrieves a pseudorandom subset of blocks from the server, and *decrypts all of them*. Of these, the blocks that actually contain data from this file are identified from each block's header. Since decryption is the most computationally intensive operation, if we can avoid decrypting the blocks not belonging to the file being read, we can speed up the operation by a constant factor (namely, α, a parameter discussed later). This is indeed possible by storing the relevant information in the first block of the file. Note that we still need to sequentially decrypt a few blocks (for our choice of parameters, up to four blocks, in expectation) before the first block of the file is encountered.
- Almost all our operations — especially the computationally intensive parts involving encryption and decryption — are "embarrassingly parallel." For instance, a set of blocks received from the server can be decrypted in parallel and assembled together using an array pre-allocated to hold all the blocks in the file.
- Our construction can be easily extended to meet a stronger security requirement, that the server does not learn the *kind of operation* (read, write or update) performed by the client (beyond what it can infer from the access pattern). For this, we shall use the update operation in place of every operation, since it offers the facility of reading and writing. (If this is used for actual updates — which allow read and write in the same operation — and if the data being written depends on the date being read, then care should be taken to avoid observable delays that can lead to a timing attack.)

C. Security Analysis

We sketch a proof of security that our construction is a secure realization of the deal blind storage functionality $\mathcal{F}_{\text{STORE}}$, for the adversary model in which the server is corrupted only passively. The proof follows the standard real/ideal paradigm in cryptography (see [12], for instance), and uses some of the standard conventions and terminology.

Roughly, the proof involves demonstrating a simulator $\mathcal{S}$ which interacts with a client only via the ideal functionality $\mathcal{F}_{\text{STORE}}$ (the ideal experiment), yet can simulate the view of the server in an actual interaction with the client in an instance of our scheme (the real experiment). The simulated view would be indistinguishable from the real view of the server, even when combined with the inputs to the client. Further — and this is the adaptive nature of our security guarantee — the inputs to the client at any point in either experiment can be arbitrarily influenced by the view of the server till then.

Before describing our simulator, we describe the main reason for security. Suppose the client makes a read access to a file f for the first time. In the *ideal experiment*, the server learns this file's size from $\mathcal{F}_{\text{STORE}}$, and nothing about the other files. In the real experiment, the server sees one or two downloads from D— a set of κ blocks S_f^0 and a set of blocks $S_\mathsf{f} \setminus S_\mathsf{f}^0$ (with the possibility that $S_\mathsf{f} = S_\mathsf{f}^0$, in which case there is only one download). Thanks to the encryption, it is easy to enforce that the *contents* of these downloaded blocks give virtually no information to the server (beyond the size of f). But we need to ensure that the *location* of these blocks also do not reveal anything more. For instance, it should not reveal how many other files are present in the system. In our construction, this is ensured by the fact that the pseudorandom subsets S_f^0 and S_f are *determined by a process that is independent of the*

The simulator $\mathcal{S}$ interacts with the functionality $\mathcal{F}_{\text{STORE}}$ on the one hand, and interacts with the server on the other, translating each message it receives from $\mathcal{F}_{\text{STORE}}$ into a set of simulated messages in the interaction between the client and the server in our scheme.

1) When it receives the initial message from $\mathcal{F}_{\text{STORE}}$ with the system parameters, $\mathcal{S}$ can calculate the size of D; it simulates the contents of the blocks in D by picking uniformly random bit strings, with the version number in each block set to 0.
2) $\mathcal{S}$ initializes a map with entries of the form $(j; \Lambda_j, \mathsf{size}_j)$, which maps an integer j (indicating the sequence number of accesses) to a sequence of blocks in D and the size of the file accessed (in blocks).
   The maps are initialized to be empty, and is filled up as $\mathcal{F}_{\text{STORE}}$ reports file accesses to $\mathcal{S}$.
3) For access number j^*, first the table entry $(j^*; \Lambda_{j^*}, \mathsf{size}_{j^*})$ is created as described below.
   Let the triple reported by $\mathcal{F}_{\text{STORE}}$ to $\mathcal{S}$ for access number j^* be $(\mathsf{op}, j, \mathsf{size})$. Recall that if $j > 0$, then the file being accessed has already been accessed (as the j^{th} access).
   a) If $\mathsf{op} = \mathsf{delete}$, then let $\mathsf{size}_{j^*} = 0$. Else, set $\mathsf{size}_{j^*} = \mathsf{size}$. Let $\ell = \max(\lceil \alpha \cdot \mathsf{size}_{j^*} \rceil, \kappa)$.
   b) If $j = 0$, then $\mathcal{S}$ samples a random sequence of ℓ distinct integers in the range $[n_{\mathsf{D}}]$, uniformly randomly, and sets Λ_{j^*} to be this sequence.
   c) Else $(j > 0)$, if $|\Lambda_j| \geq \ell$, set $\Lambda_{j^*} = \Lambda_j$; else $(j > 0$, and $|\Lambda_j| < \ell)$, extend Λ_j to a sequence of length ℓ uniformly at random (without duplicates). Set Λ_{j^*} to be this extended sequence.
4) Next, $\mathcal{S}$ creates the simulated view in which first the server gets a request to download κ blocks indexed by the first κ entries of Λ_{j^*}; if $\ell > \kappa$, this is followed by a request to download blocks indexed by the next $\ell - \kappa$ entries of Λ_{j^*}. For operations other than read, this is followed by an upload consisting of new versions (with the blocks' version numbers incremented, and with fresh random strings as contents) of the blocks indexed by the first ℓ entries of Λ_{j^*}.

Fig. 5: Description of the simulator $\mathcal{S}$ used in the proof of Theorem 1.

other files in the system – they are chosen randomly (or rather, pseudorandomly) for each file independently. The other files in the system influence the subset $\widehat{S}_{\mathsf{f}} \subseteq S_{\mathsf{f}}$ of blocks that actually carries the data (because these blocks must not be shared with the data-carrying blocks of any other file). However, due to the encryption, the server does not learn anything about $\widehat{S}_{\mathsf{f}}$ (beyond the fact that it must be a subset of S_{f}).

Formally, a simulator can simulate the view of the adversary *randomly*, based only on the size of the file f being accessed. The only difference between this simulation and the real execution (beyond what is hidden by the encryption and the security of pseudorandomness) is the following: in the real execution, there is a small probability that an update could fail, if there are not enough free blocks within the pseudorandom subset S^0_{f} or S_{f}. In the simulation, no failure occurs. Thus the crucial argument in proving security is to show that it is only with negligible probability that the client would be left without adequate number of free blocks in such a pseudorandom set, forcing it to abort the protocol. We will give a standard probabilistic argument to prove that this is indeed the case.

In the proof below we describe our simulator $\mathcal{S}$ more formally, and then discuss the main combinatorial argument used to show that the simulation is indistinguishable from the real execution. For the sake of clarity, we leave out some of the routine details of this proof, and focus on aspects specific to our construction.

The following theorem statement is in terms of the "*storage slack ratio*" in a Blind Storage system, which is the ratio of the number of blocks n_{D} in the system to the number of blocks of (formatted) data in the files stored in the system. Note that the storage slack ratio decreases as files are added (or updated to become longer) and increases as files are deleted (or updated to become shorter). The security guarantee below uses the standard security definition in cryptography literature (see, for instance, [12]), which assures that the security "error" (statistical distance between the simulated execution and the real execution) is *negligible*,[8] as a function of the security parameter. Later, we discuss the choice of concrete parameters.

Theorem 1: *Protocol* SCATTERSTORE *securely realizes the functionality $\mathcal{F}_{\text{STORE}}$ against honest-but-curious adversaries, provided the storage slack ratio at all times is at least $\frac{2}{1-1/\alpha}$ and $n_{\mathsf{D}} \geq \kappa = \omega(\log k)$.*

Proof: The non-trivial case is when the server is corrupt (honest-but-curious) and the client is honest. We describe a simulator for this setting in Figure 5. The simulator essentially maintains the indices of the sets of blocks seen by the server. It need not maintain the subsets within these sets that carry actual data for the file being accessed. The maps are used to maintain consistency in terms of the pattern (same subsets are used if the same file is accessed) and the size of the files.

There are two differences between this simulation and the real execution. Firstly, the simulated execution uses truly random strings instead of the outputs from Φ, Φ and Γ. To handle this we can consider a "hybrid experiment" in which the real execution is modified so that instead of Φ, Ψ and Γ, truly random functions are used. By the security guarantees of the PRF, the FD-PRF and the PRG (applied one after the other), this causes only an indistinguishable difference.

The second difference is in aborting: in the real protocol, the client aborts when it cannot find enough free blocks in a pseudorandom subset, whereas the simulation never aborts. Conditioned on the protocol never aborting in the hybrid execution, the server's view in that execution is identical to that in the simulated execution.

To complete our proof, therefore it remains to show that the probability of the client aborting in the hybrid (or real) protocol is negligible. We denote this probability by p_{err}. Before proceeding, we remark that our goal here is to give an asymptotic proof of security (showing that p_{err} goes down as a negligible function of the security parameter). The concrete

[8] A function $\nu : \mathbb{N} \to \mathbb{R}^+$ is said to be negligible if, for every $c > 0$, there exists a sufficiently large $k_0 \in \mathbb{N}$ such that for all $k \geq k_0$, $\nu(k) < \frac{1}{k^c}$. That is, $\nu(k)$ becomes smaller than $1/\text{poly}(k)$ eventually, for any polynomial poly.

parameters from this analysis are overly pessimistic and an actual implementation can use less conservative parameters. The key message is that p_{err} provides a bound on the extent of insecurity, and this probability can be quickly driven down by modestly large parameters that scale linearly with the size of the data stored.

To analyze p_{err}, recall that we are analyzing a modified execution in which the output of the PRG Γ on pseudorandom seeds (used to define the pseudorandom subsets) have been replaced with truly random strings. Suppose there has been no abort so far, and a new file f of size_f blocks is to be inserted into the system (either during the BSTORE.Build stage of during an update or write operation). Let d out of the n_D blocks in D be filled. These blocks were filled by picking random subsets, and then within these subsets, choosing random subsets with free blocks. The net effect is of choosing a random subset of d blocks out of the n_D blocks. Now, when f is being inserted, we pick a random subset S^0_f of size κ and a random set $S_\mathsf{f} \supseteq S^0_\mathsf{f}$ of size $|S_\mathsf{f}| = \max(\lceil \alpha \cdot \mathsf{size}_\mathsf{f} \rceil, \kappa)$. The *expected number* of occupied blocks within this set is $\frac{d}{n_\mathsf{D}} \cdot |S_\mathsf{f}|$. By a standard application of Chernoff bound,[9] the probability that more than $2\frac{d}{n_\mathsf{D}}|S_\mathsf{f}|$ blocks are occupied is $2^{-\Omega(|S_\mathsf{f}|)}$, provided $\frac{d}{n_\mathsf{D}}$ is upperbounded by a constant less than 1. Since $|S_\mathsf{f}| \ge \kappa$, this probability is $2^{-\Omega(\kappa)}$, and since κ is super-logarithmic in k (for e.g., $\log^2 k$), this probability is $2^{-\omega(\log k)}$ which is negligible in k. Thus except with negligible probability, of the $|S_\mathsf{f}|$ blocks chosen, at least $|S_\mathsf{f}|(1-2\frac{d}{n_\mathsf{D}}) \ge \alpha \cdot \mathsf{size}_\mathsf{f} \cdot (1-2\frac{d}{n_\mathsf{D}})$ are free.

By the hypothesis in the theorem statement, the storage slack ratio $\frac{n_\mathsf{D}}{d} \ge \frac{2}{1-1/\alpha}$, or equivalently, $1-2\frac{d}{n_\mathsf{D}} \ge \frac{1}{\alpha}$. Thus, except with negligible probability, of the $|S_\mathsf{f}|$ blocks chosen $\alpha \cdot \mathsf{size}_\mathsf{f} \cdot (1-2\frac{d}{n_\mathsf{D}}) \ge \mathsf{size}_\mathsf{f}$ blocks are free. The same analysis shows that S^0_f will have at least one free block (in fact, at least $\lfloor \kappa/\alpha \rfloor$ free blocks), except with negligible probability. If both these conditions hold, the client will not abort when adding this file. By a union bound, the probability that it aborts remains negligible as long as it adds only polynomially many files. ■

On the choice of parameters. There are a few parameters that one can set in an implementation of our blind storage scheme to optimize security levels and performance. For simplicity we treat p_{err} (which measures the probability that any *illegitimate* information is revealed to the server) as fixed at either 2^{-40} or 2^{-80}. The other important parameters are the following:

- γ, an upperbound on the storage slack ratio — i.e., $\frac{n_\mathsf{D}}{d_0}$, where d_0 is an upperbound on the total number of blocks of all the files (formatted correctly);
- α, the ratio between the number of blocks in a (large enough) file and the number of blocks in the pseudorandom subset which is downloaded/uploaded when that file is accessed; and
- κ, the minimum number of blocks in a pseudorandom subset.

The higher these parameters, the better the security level would be. However, they also reflect higher storage and communication costs. One can find different combinations of (γ, α, κ) to meet a security level (probability of "error" in simulation) using the following explicit upperbound, which is tighter than the Chernoff bound used for asymptotic analysis above.[10]

$$p_{\text{err}}(\gamma,\alpha,\kappa) \le \max_{n \ge \frac{\kappa}{\alpha}} \sum_{i=0}^{n-1} \binom{\lceil \alpha n \rceil}{i} \left(\frac{\gamma-1}{\gamma}\right)^{i} \left(\frac{1}{\gamma}\right)^{\lceil \alpha n \rceil - i}$$

Figure 6 plots various possible combinations of α and κ for various choices of p_{err} and γ. A few suggested choices of (γ, α, κ) which achieve $p_{\text{err}} \le 2^{-40}$ are $(4, 4, 45)$, $(2, 8, 60)$ and $(4, 8, 25)$. Thus, for instance, one could use the parameter setting of $(\gamma, \alpha, \kappa) = (4, 4, 45)$ which means that the amortized storage requirement for each file and the communication requirement for reading *large files* is roughly 4 times the size of the file; however, for small files – any file with at most 11 blocks, including empty or non-existent files – 45 blocks would be downloaded, decrypted, reencrypted and uploaded back.

While a very large value of κ would require a large amount of communication and extra computation on part of the client (for updates), we recommend moderately large values for κ. This is because, firstly, *increasing κ does not have any effect on the storage needed* (because only as many blocks are occupied in D as the actual data consists of), and secondly it actually provides a higher security guarantee and may slightly increase the overall efficiency too! Apart from lowering p_{err}, another reason for a higher security guarantee (not captured in $\mathcal{F}_{\text{STORE}}$, for simplicity) is that the server does not learn the exact number of blocks in every file that is accessed; for "small" files, it learns only that the file is small (at most 11 blocks, in the above example). The higher the value of κ, the less the information that the server learns. The potential (slightly) higher efficiency is due to the fact that when a "small" file is retrieved, a single round of interaction suffices, and again, the higher the value of κ, the more the files that fall into the "small" category. This does not increase the computational cost during read operations.

We point out that while α, κ (and n_D) are parameters built into the system specification, it is not necessary to have a hard bound d_0 on the number of blocks of D that can be filled. In other words, γ and p_{err} exhibit graceful degradation: as the array D fills up and γ decreases, p_{err} increases.

Another parameter that affects the choice of these parameters is the size of the blocks in D. As the block size decreases, on the one hand, the number of blocks in files grows and the effect of the communication overhead due to the minimum number of blocks used for small files (the parameter

[9] In choosing a random subset of blocks, the blocks are not chosen independent of each other. So in order to apply Chernoff bound, we first consider the experiment in which the blocks are selected independent of each other with the same fixed probability, so that the expected number of blocks chosen is, say $3/2d$. Then, by an application of Chernoff bound, except with $2^{-\Omega(n_\mathsf{D})}$ probability, at least d blocks are occupied. Now, in this experiment, we bound the probability that more than $2\frac{d}{n_\mathsf{D}}|S_\mathsf{f}|$ blocks in S_f, again using Chernoff bound. This probability is an upperbound on the corresponding probability in the original experiment.

[10] The error probability when adding a file of $n \ge \frac{\kappa}{\alpha}$ blocks is upperbounded by the probability that when $\lceil \alpha n \rceil$ blocks are picked (with replacements) from a set of n_D blocks of which at most d_0 would be occupied, $i < n$ distinct blocks that are picked are free. The actual experiment involves picking blocks without replacement, but for our range of parameters, this gives a valid upperbound.

κ) decreases; on the other hand, the overhead due to the header size in each block increases.

An implementation can choose a default standard setting of the above parameters, or seek to optimize performance by tuning them to suit the profile of the files to be stored in the system. For instance, the set of parameters appropriate for an application like our SSE construction in the sequel (in which the keyword index files are stored in a Blind Storage system) may be different from those appropriate for an application storing a relatively small number of large files. But it is important that any such optimization is based on a *public* profile of the set of files to be stored in the system. This is because, conservatively, we should assume that the server would know all the system parameters (and exact sizes of the files accessed). It is true that, heuristically, slightly better guarantees may be available, since the server learns only $\max(\lceil \alpha \cdot \mathsf{size} \rceil, \kappa)$, and need not exactly know α and κ (except as revealed by the former, combined with any auxiliary information it may have about the sizes of the files being accessed). Further heuristics could be employed to make this information noisy, so that it remains hard to decipher the parameters even from a large number of correlated accesses. Nevertheless, we recommend that the system parameters are optimized only using information that can be made known to the server.

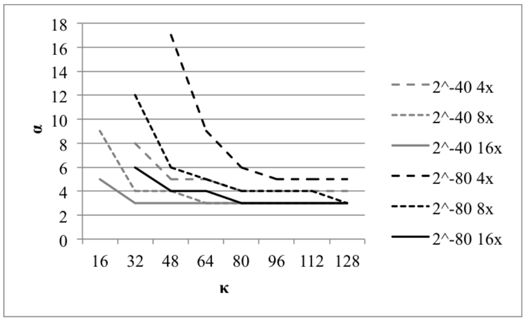

Fig. 6: Finding the right parameters. Each line on the graph corresponds to trade-offs between α and κ for a choice of $p_{\text{err}} \in \{2^{-40}, 2^{-80}\}$ and $\gamma \in \{4, 8, 16\}$.

V. Searchable Symmetric Encryption

In this section we formally define the syntax and security requirements of a dynamic SSE scheme, and also present an efficient construction. As we shall see, our syntax for a dynamic SSE scheme is simpler than in [18], since all non-trivial operations are carried out by the client, and hence, there are no server side algorithms to be specified. Our construction

A. *Definitions*

A dynamic searchable symmetric encryption scheme (or simply, SSE) consists of five probabilistic polynomial time procedures (run by the client), $\mathsf{SSE.keygen}$, $\mathsf{SSE.indexgen}$, $\mathsf{SSE.search}$, $\mathsf{SSE.add}$ and $\mathsf{SSE.remove}$. These procedures interact with a "dumb" server which provides download and upload facilities to access blocks in an array (see Section IV-A), and also a simple file-system to lookup documents by identifiers. Looking ahead, in our implementation, the upload and download facilities are used to implement a blind-storage scheme which is used to store the keyword indices, and the file lookup facility is used to store the actual (encrypted) documents.[11]

- $\mathsf{SSE.keygen}$: Takes the security parameter as input, and outputs a key K_{SSE}. All of the following procedures take K_{SSE} as an input.
- $\mathsf{SSE.indexgen}$: Takes as input the collection of all the documents (labeled using document IDs), a dictionary of all the keywords, and for each keyword, an *index file* listing the document IDs in which that keyword is present.[12] It interacts with the server to create a representation of this data on the server side.[13]
- $\mathsf{SSE.search}$: Takes as input a keyword w, interacts with the server, and returns all the documents containing w.
- $\mathsf{SSE.add}$: Takes as input a new document (labeled by a document ID that is currently not in the document collection), interacts with the server, and incorporates it into the document collection.
- $\mathsf{SSE.remove}$: Takes as input a document ID, interacts with the server, and if a document with that ID is present in the server, removes it from the document collection.

Security Requirement. As in the case of blind-storage, we specify an ideal functionality, $\mathcal{F}_{\text{SSE}}$ (Figure 7) to capture the security requirements of a dynamic SSE scheme. We note that the standard simulation-based security (with an environment) applied to the functionality $\mathcal{F}_{\text{SSE}}$ automatically ensures what has been called security against adaptive chosen keyword attacks (CKA2-security) for searchable encryption.

The functionality $\mathcal{F}_{\text{SSE}}$ is described in detail in Figure 7. If document-set privacy is required, then the functionality behaves slightly differently: the original set of documents are not added to the list of documents Δ upfront, but each one is added only at the first instance when it is accessed using $\mathcal{F}_{\text{SSE}}.\mathsf{search}$ or $\mathcal{F}_{\text{SSE}}.\mathsf{remove}$.

We highlight a few aspects of our security definition, compared to that in [18] and prior work. In all forms of SSE, *keyword access pattern* and *document access pattern* are revealed: i.e., if the same keyword is searched for multiple times or if the same document appears in multiple keyword searches, the server learns about that; techniques for hiding this information incur significant costs. The goal of an SSE scheme is to reveal as little information as possible, beyond this information. In our scheme we reveal very little information beyond this, for the *original set of documents*. For newly added documents, a little more information is revealed, as they are added (see *Addition leakage* in Figure 7). This has the effect that for every subset of newly added documents, the server learns only the *number of keywords that are common to all the documents in that subset*.

Existing schemes reveal significantly more information. For example, in [18], when a document is removed, the scheme reveals *the number of keywords in the document* and further,

[11] In our scheme, if we opt to have document-set security, then the file lookup facility is not used, as the documents will also be stored in the blind-storage.

[12] The index files can be created by $\mathsf{SSE.indexgen}$, if it is not given as input.

[13] Typically, this would consist of a collection of (encrypted) documents, labeled by document indices (different from document IDs), and a representation of the index, which in our constructions will be stored using a blind-storage system.

- *Initialization.* On receiving the command $\mathcal{F}_{\text{SSE}}.\mathsf{indexgen}$ from the client, $\mathcal{F}_{\text{SSE}}$ accepts a set of ∂_0 documents — refered to as the *original documents* (as opposed to newly added documents) — and stores them internally in an array Δ. For $1 \leq \partial \leq \partial_0$, the array stores $\Delta[\partial] = (\mathsf{id}_\partial, \mathsf{contents}_\partial, \mathsf{W}_\partial)$ — a unique document ID, the document contents and a set of keywords in the document. It also accepts from the client a number N, which is a (possibly liberal) upperbound on the total number of (keyword, document) pairs that will be present in the system at any one time. $\mathcal{F}_{\text{SSE}}$ also maintains a set called $\mathsf{Removed}$ initialized to $\emptyset$.
  - **Initialization Leakage.** $\mathcal{F}_{\text{SSE}}$ reveals to the server the $(\mathsf{N}, s_1, \cdots, s_{\partial_0})$ where $s_\partial = |\mathsf{contents}_\partial|$ (in number of bits).
- *Addition.* On receiving the command $\mathcal{F}_{\text{SSE}}.\mathsf{add}$ to add a document $(\mathsf{id}, \mathsf{contents}, \mathsf{W})$ (with a new or existing document ID), $\mathcal{F}_{\text{SSE}}$ appends it to the array Δ: i.e., if there are $\partial - 1$ entries currently, let $\Delta[\partial] = (\mathsf{id}, \mathsf{contents}, \mathsf{W})$. If there exists $\partial' < \partial$ with $\mathsf{id}_{\partial'} = \mathsf{id}$ and $\partial' \notin \mathsf{Removed}$, then add ∂' to $\mathsf{Removed}$.
  - **Addition Leakage.** $\mathcal{F}_{\text{SSE}}$ reveals to the server the updated set $\mathsf{Removed}$ and $\{M_w^{\mathsf{new}} | w \in \mathsf{W}\}$, where $M_w^{\mathsf{new}} = \{\partial' | \partial' > \partial_0 \text{ and } w \in \mathsf{W}_{\partial'}\}$. i.e., M_w^{new} is the set of *newly added documents* (i.e., not the original documents) that have the keyword w. Note that only the sets M_w^{new}, and not their labels w, are shared with the server.
- *Removal.* On receiving the command $\mathcal{F}_{\text{SSE}}.\mathsf{remove}$, $\mathcal{F}_{\text{SSE}}$ accepts a document ID id and identifies ∂ (if any) such that $\mathsf{id}_\partial = \mathsf{id}$ and $\partial \notin \mathsf{Removed}$. If such an index ∂ exists, it adds ∂ to $\mathsf{Removed}$.
  - **Removal Leakage.** $\mathcal{F}_{\text{SSE}}$ reveals to the server the updated set $\mathsf{Removed}$.
- *Search.* On receiving the command $\mathcal{F}_{\text{SSE}}.\mathsf{search}$, it accepts a keyword w from the client and returns $\{(\mathsf{id}_\partial, \mathsf{contents}_\partial) | w \in \mathsf{W}_\partial \text{ and } \partial \notin \mathsf{Removed}\}$ to the client.
  - **Search Leakage.** $\mathcal{F}_{\text{SSE}}$ reveals to the server the last instance the same keyword was searched on (or that it is being searched for the first time) and also $M_w = \{\partial | w \in \mathsf{W}_\partial\}$.

Fig. 7: The $\mathcal{F}_{\text{SSE}}$ functionality: all the information leaked to the server in our SSE scheme is specified here.

for each keyword in it, *up to two other documents that share the same keyword*. This is the case even if that keyword is never searched on. In contrast, by our security requirement, if an original document is removed, only the number of *keywords in it that are searched* can be revealed. Further, it is not revealed that a removed document shared a keyword with another document, unless such a keyword is explicitly searched for.

We remark that our functionality reveals "removed" versions of the documents in search results, but this information was revealed (implicitly) by the leakage functions in [18] as well, as the identifiers for each keyword in a removed document is revealed and this information links the removed documents to future searches on the same keyword (when the same identifier for the keyword is revealed).

Finally, our scheme allows the client to refer to a document using an arbitrary document ID rather than a serial number (which is useful when removing documents from the collection). We also allow the client to reuse document IDs. The server does learn when a document ID is reused (though not the actual identifier of the document ID itself); further, in the pattern information revealed to the server, the different versions that use the same document ID are differentiated. Other dynamic SSE schemes often avoid this aspect simply by not using document IDs. This suffices if the only time a document is removed is immediately after retrieving it from a search (or if the client is willing to maintain a map from document IDs to serial numbers); however, realistically, in many applications of a dynamic SSE scheme, it will be important to efficiently remove documents referenced by their document IDs.

B. Searchable Encryption from Blind Storage

In this section, we describe an efficient dynamic searchable encryption scheme, BSTORE-SSE, built on top of a blind-storage scheme. The full details are given in Figure 8. Here we sketch the main ideas.

First, note that we can implement a *static* searchable encryption scheme simply by storing the index file for each keyword (which lists all the documents containing that keyword) in a blind storage system. The guarantees of blind storage readily translate to the security guarantees of searchable encryption: the server learns only the pattern of index files (i.e., keywords) accessed by the client.

In a dynamic searchable encryption scheme, we need to support adding and removing documents, which in turn results in changing the index files. We seek to do this without revealing much information about the keywords in a document being added or removed, if those keywords have not been searched on before. To support dynamic searchable encryption (with much better security guarantees than previous constructions), we rely on the following observation. The access pattern that server would be allowed to learn tells the server if two newly added documents share a keyword or not, *as soon as they are added and before such a keyword is searched for* (but not whether they share keywords with the original set of documents that were added when initializing the system). This means we can treat the set of newly added documents virtually as a different system, with significantly weaker security requirements.

Thus, for each keyword, we use two index files: one listing the original documents that include that keyword, and another listing the newly added documents that include it. The first index file is stored with the server using a blind storage scheme, where as the second can be stored in a "clear storage" system (see below). Searching for keywords now involves retrieving both these index files. Adding documents involves updating only the second kind of index files (using an append operation of the clear storage). Also, removing a newly added document involves updating only the second kind of index files, which is straightforward (except for efficiency concerns, addressed below). But in removing an original document, we need to ensure that the information on keywords in it that are not searched for (for e.g., the number of such keywords) remains secret. This is achieved by a *lazy deletion* strategy. The index file of a keyword (for the original set of documents) is not updated until that keyword is searched for. At that point, if the client learns that a document listed in that index has been deleted, the index is updated accordingly. This update can be

The construction uses a blind-storage system BSTORE, and a pseudorandom permutation Ψ' for mapping document IDs (with versioning) to pseudorandom document indices. It also uses a clear-storage system CLEARSTORE (see text).

- SSE.keygen: Let $\mathsf{K}_{\text{SSE}} = (\mathsf{K}_{\text{BSTORE}}, \mathsf{K}_{\partial\mathsf{ID}})$ where $\mathsf{K}_{\text{BSTORE}}$ is generated by BSTORE.Keygen and $\mathsf{K}_{\partial\mathsf{ID}}$ is a key for the PRP Ψ'.
- SSE.indexgen:
  1) Firstly, for each document ∂, assign a pseudorandom ID $\eta_\partial = \Psi'_{\mathsf{K}_{\partial\mathsf{ID}}}(\mathsf{id}_\partial)$, where id_∂ is the document ID.
  2) For each keyword w that appears in at least one document, construct an *index file* with file-ID index_w that contains η_∂ for each document ∂ that contains the keyword w. No specific format is required for the data in this file; in particular, it could contain a "thumbnail" (of fixed size) about each document in the list.
  3) Next, initialize a blind-storage system with the collection of all these index files (using BSTORE.Build).
  4) Also, (outside of the blind-storage system) upload encryptions of all the documents labeled with their pseudorandom document index η_∂.
- SSE.remove: To minimize the amount of information leaked, and for efficiency purposes, we rely on a lazy delete strategy.
  1) Given a document ID id_∂, check if a document with index $\eta_\partial = \Psi'_{\mathsf{K}_{\partial\mathsf{ID}}}(\mathsf{id}_\partial)$ exists, and if so remove it, using the file system interface of the server. The index files (in the blind storage or the clear storage) are not updated for the keywords in this document right away, but only during a subsequent search operation (see below).
- SSE.add: To add a document ∂ to the document collection, first call SSE.remove to remove any earlier copy of a document with the same document ID. Then proceed as follows:
  1) Compute a pseudorandom document index $\eta_\partial = \Psi'_{\mathsf{K}_{\partial\mathsf{ID}}}(\mathsf{id}_\partial)$.
  2) Generate a random tag tag and add it to the document (say, as a prefix, before or after encrypting the document). Encrypt the document and upload it, as in the SSE.indexgen phase, using the label η_∂.
  3) Then, for each keyword w that appears in this document, use the append facility of the clear-storage scheme to append a record consisting of $(\eta_\partial, \mathsf{tag})$ to the file with file-ID index_w to include η_∂. Note that the append operation will create a file in the clear storage system, if it does not already exist.
- SSE.search: Given a keyword w, retrieve and update the index files with file-ID index_w and index_w as follows:
  1) Retrieve the index file index_w from the blind storage system using the first stage of update operation of the blind storage scheme. Also, retrieve the index file index_w from the clear storage system, using the first stage of its update operation. All the documents containing the keyword w have their document indices listed in these two index files. Attempt to retrieve all these documents listed from the server.
  2) Some of the documents listed in the index file index_w could have been removed. Complete the blind storage update operation on the file index_w to erase the removed files from its list, without changing the size of the file.
  3) Some of the documents listed in the index file index_w may have been removed or replaced with newer versions. Complete the clear storage update operation on the file index_w to remove from its list any document that could not be retrieved, or for which the listed tag did not match the one in the retrieved document. (Both the update operations are completed in the same round).

  One could add an extra round to first check just the tags of the documents before retrieving the documents themselves.

Fig. 8: Searchable Encryption Scheme BSTORE-SSE

carried out in a single update operation of the blind storage scheme, with little overhead.

In fact, for removing newly added documents too, we follow a similar lazy delete strategy, for efficiency purposes. (Otherwise, during a delete operation, the client will need to fetch the index files for all the keywords in the deleted document in order to update them, unless the server is willing to carry out a small amount of computation.) However, we need to account for the possibility that a document ID could be reused and that a later version may not have a keyword present in an earlier version. We associate a random tag with a document to check if the version listed in an index is the same as the current version.

Properly instantiated, this simple idea yields strictly better security than prior dynamic searchable encryption schemes [18], [10] which revealed more information about keywords not searched for, especially when removing documents.

Clear Storage. To store the index files for newly added documents, our SSE scheme uses a "clear storage" scheme CLEARSTORE that supports the following operations:

- Files labeled with file-IDs can be stored (in the clear, without any encryption). A two-stage update operation can be used to read this file and then write back an updated version (which could be shorter).
- In addition, there is an efficient append operation, that allows appending a record (of fixed size) to the file in constant time (without having to retrieve the entire file and update it).

Note that a standard file-system interface provided by the server can support all these operations. But the append operation may not be supported by a cloud storage provider. In this case, it can be implemented by the client, as we consider in our evaluation.

We consider a simplified version of the SCATTERSTORE to implement CLEARSTORE with efficient append. In this implementation, to store a file $\mathsf{f} = (\mathsf{id}_\mathsf{f}, \mathsf{data}_\mathsf{f})$, the file data data_f is stored (unencrypted) in a subset of blocks of a pseudorandom set $\widehat{S}_\mathsf{f} \subseteq S_\mathsf{f}$. We use a separate file-system interface (without append) to store fixed-size header files labeled with the file-name id_f; This header file stores an index indicating the *last block* of S_f that is occupied by the file (i.e., the length of the shortest prefix of S_f that contains $\widehat{S}_\mathsf{f}$[14] To append a record to a file, the client retrieves the header block via the file-system interface, using the file-name id_f. Then it generates S_f, and recovers the i^{th} block in S_f, where i is the index stored in the header block. Then it checks if there is enough space in this

[14] The first block of the data could also be stored in the header file. Note that then it is possible that the header block itself contains all the data of the file; in this case the index indicating the last block is set to 0.

last block, and if so adds the record there. Else, it generates κ more entries in S_{f}, fetches those blocks from the clear storage, adds the record to the first empty block in this sequence,[15] and updates the index of the last block stored in the header file accordingly. Note that the number of blocks fetched is a constant on average provided a block is large enough to contain (say) κ records; the number of blocks written back is at most two (and on the average, close to 1).

Choice of parameters. We instantiate BSTORE-SSE with our SCATTERSTORE constructions. By choosing the parameter κ for SCATTERSTORE, we can ensure that a single search operation can typically be completed in one and half rounds of interaction. This is because the typical size of an index file could fit into a few blocks, and by choosing $\kappa = 80$ as we do in our experiments, the index file can often be retrieved without having to fetch more blocks. However, in the worst case (e.g., searching for the keyword "the," as we report), two and half rounds of interaction will be needed.

Theorem 2: *Protocol* BSTORE-SSE *securely realizes* $\mathcal{F}_{\text{SSE}}$ *against honest-but-curious adversaries.*

Proof Sketch: The security of this scheme is fairly straightforward to establish, since it uses the blind storage scheme as a blackbox, and involves no other cryptographic primitive. All the information available to the server from the blind storage scheme as used in this construction (i.e., the access patterns of the index files) is easily derived from the information that the server is allowed to have in the searchable encryption scheme. In other words, a simulator can simulate to the server all the messages in the protocol using the information it obtains in the ideal world. The details are straightforward, and hence omitted. □

VI. IMPLEMENTATION DETAILS

We implemented prototypes of our blind storage and searchable encryption schemes. The code was written in C++ using open-source libraries. We used Crypto++ [1] for the block cipher (AES) and collision-resistant hash function (SHA256) implementations.

As our schemes only require upload and download interface and do not require any computation to be performed on the server, they can be implemented on commercially available cloud storage services. As a proof of concept, we further implemented a C++ API to interface with Dropbox's Python API. This enables a Dropbox user to use a C++ implementation of BSTORE-SSE (using SCATTERSTORE) with Dropbox as the server. In our Dropbox implementation, each *block* in the SCATTERSTORE scheme is kept as a file in Dropbox. We recommend using SCATTERSTORE with a block size that is a multiple of the block size in the cloud storage provider's storage (typically, 4KB).

VII. SEARCHABLE ENCRYPTION EVALUATION

For concreteness, we will compare the performance of our SSE scheme with that of the recent scheme in [18], as one of the most efficient dynamic SSE schemes in the literature, implemented in a comparable setting. The more recent work of [6] offers a possibly more optimized version of this protocol (without dynamic functionality), but is harder to compare against experimentally, as the reported implementation was in a high performance computing environment. We remark that for the case of simple keyword searches (which is not the focus of [6]), the construction of [6] is similar to that of [10], [18], and is expected to show similar performance.

We focus on computational costs; space and communication overheads in the prior constructions are often not reported making a direct comparison hard.

- The computation times reported are for the client. In our case the server is devoid of any computation (beyond simple storage tasks) and hence this constitutes all the computation in the system. In contrast, in previous SSE schemes, the server's computation is often much more than that of the client. Thus it would already be a significant improvement if our client computation costs are comparable to the client computation costs in prior work. As we shall see, this is indeed the case.
- There are several possible engineering optimizations in the Blind-Storage scheme which can significantly improve the performance of the SSE scheme (for instance, the first one listed in Section IV-B3 cuts down the time taken for the search operation by a factor of α or more). *None of these optimizations* have been implemented in the prototype used for evaluation.

Datasets. We use two datasets to evaluate our searchable encryption scheme, *emails* and *documents*.

1) For emails we use the Enron dataset [2] which was also used by [18] and several other works. From the Enron e-mail dataset, we selected a 256MB subset, consisting of about 383,000 unique keywords and 20,695,000 unique (document, keyword) pairs. In the experiments involving smaller amounts of data, subsets of appropriate sizes were derived from these datasets.

2) For documents, we created a dataset with 1GB of four types of documents, namely PDF, Microsoft PowerPoint, Microsoft Word and Microsoft Excel. The documents were obtained by searching for English language documents with filetypes *pdf*, *ppt*, *doc* and *xls*, using Google search. The resulting collection consists of 1556 documents (roughly evenly distributed among the four filetypes), with over 214,000 unique keywords and about 1,372,000 unique (document, keyword) pairs.

Experiments. The code was compiled without any optimizations on Apple Mac OS X. We used a well provisioned laptop – with Intel Core i7 3615QM processor, 8GB memory, running Mac OS X 10.9 – for the experiments, keeping in mind that the typical user of our system will use searchable encryption on a cloud via her personal computer, just the same way a cloud storage service like Dropbox is used. This is in contrast with prior research which typically evaluated their work on large servers with large amounts of memory.

As we shall see below, our scheme is highly scalable and practically efficient. We cannot offer a direct comparison between our performance speeds and that of [18], because of

[15] Unlike in the case of blind-storage, if no empty block is found among the blocks fetched, the client can go on to fetch more blocks. This also allows one to optimistically fetch a smaller number of blocks, without a significant penalty.

different hardware configuration and limited test equipment information presented in [18]. Nevertheless, our evaluation shows that our scheme should be significantly more efficient than that of [18].

A. Micro-benchmarks – File-keyword pair analysis

In [18], micro-benchmarks were used to evaluate the SSE operations. We do the same for the SSE.indexgen algorithm. (For our search, add and delete operations, the performance is essentially independent of the total number of file-keyword pairs already stored in the system, and this micro-benchmark does not provide a meaningful evaluation of these operations. These operations are evaluated differently, as explained below.)

Figure 9 shows micro-benchmarks for our scheme. The parameters used in the scheme are held constant, and are the same as detailed in the next section. Each data point is an average of 5 runs of SSE.indexgen. Note that the amortized per-pair time falls as the number of pairs increases, before tending to 1.58μs; this is because our SSE.indexgen operation involves encrypting the whole array D (which has the same size in all the experiments), and this overhead does not increase with the number of pairs.

Compared to the time for index generation operation reported in [18], our performance is significantly better. [18] reports a per-pair time of 35μs for the same operation. Thus our index generation operation is an order of magnitude faster.

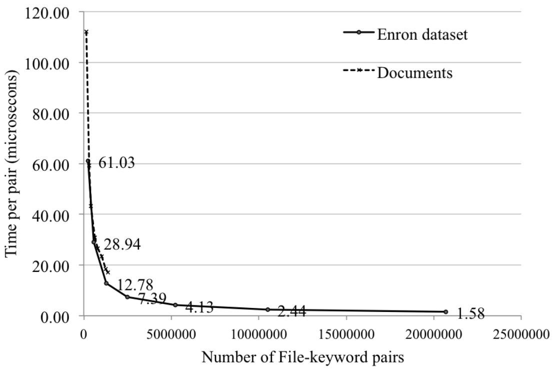

Fig. 9: File/Keyword pair versus amortized time for SSE.indexgen. Time per file/keyword pair tends to 1.58μs, much better than the 35μs reported in [18] in a similar dataset.

B. Full evaluation

Each data point for Index Generation is the average of 5 runs of SSE.indexgen. Each data point for the Search is the average of 5 runs using the most frequent English word "the". Each data point for addition is the average of at least 5 runs.

1) Parameters Used: The parameters used for the experiments guarantee $p_{\text{err}} \leq 2^{-80}$ (recall that p_{err} is the probability of the scheme aborting and measures the security "error") if less than $1/8$ of the total blocks in D are filled and guarantee $p_{\text{err}} \leq 2^{-40}$ if less than $1/4$ of the total blocks in D are filled. We set $\kappa = 80$ and $\alpha = 4$, block size of D to 256 bytes, the total number of blocks in D to $n_{\mathsf{D}} = 2^{24}$.

2) Index Generation: Index generation is computationally the most expensive phase of any searchable encryption scheme. Our index generation performance measurements include encryption of documents and all other operations except the cost of plaintext index generation. Plaintext index generation performance is orthogonal to our contributions, doesn't reflect the performance of our system and is ignored by all prior work. Figure 10 shows our index generation performance on the email dataset. Our performance is much better when compared to that of [18], which takes 52 seconds to process 16MB of data. Our scheme can process 256MB (16 times more data) in about 35s. [18] extrapolates this to to 16GB of text e-mails without any attachments and, since the time for index generation scales roughly linearly with data, estimates that their index generation would take 15 hours; in contrast, it would take only 41 minutes in our scheme.

This matches our conclusion from the micro-benchmarks evaluation, that our index generation operation is at least an order of magnitude faster than that of [18].

Figure 11 shows the performance of our scheme on the document dataset.

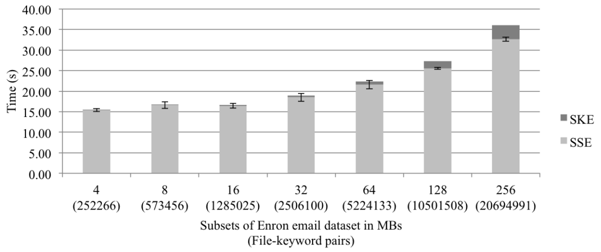

Fig. 10: SSE.indexgen performance on email dataset with 99% confidence intervals: SKE stands for Symmetric Key Encryption and is the time required to encrypt the documents. All SKE costs are non-zero but some are very small.

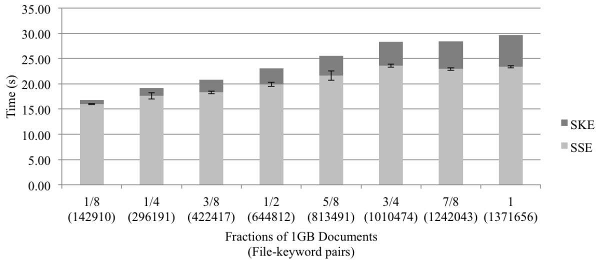

Fig. 11: SSE.indexgen on the document dataset with 99% confidence intervals

Communication costs. The communication cost of initial index upload depends upon the parameters used for Blind-Storage, and specifically, the size of the array D. As mentioned above, in Section VII-B1, the size of D was set to 1GB (2^{24} blocks of 256 bytes each) in our experiments. In comparison, the actual amount of index data for the 256MB subset of the email dataset consisted of 20,694,991 file-keyword pairs, which, using 4-byte fields for document IDs, translates to about 78MB of data. Given the small size of some of the index files, on formatting this data into 256-byte blocks for the Blind-Storage scheme, this resulted in about 178MB data. For our choice of κ and α, $\gamma = 4$ is sufficient to bring p_{err} below 2^{-40}. That is, it would be sufficient to use about 712MB as the size of D. Hence the choice of 1GB as the size of D in our experiments leaves abundant room to add more documents later.

For the document dataset, there are only 1,371,656 file-keyword pairs, which translates to a plaintext index size of 5MB (with 4-byte document IDs). Thus the size of D could be as low as 20MB. Note that the document collection itself is of size 1GB in this case. For rich data formats, it will typically be the case that the communication overhead due to SSE.indexgen would be only a fraction of the communication requirement for the documents themselves.

3) Search: Figure 12 shows the search performance of our scheme excluding the final decryption of the documents. Figure 12 does include overhead incurred at search time to handle lazy delete. We searched for the most frequent English word “the” and it was present in almost all the documents. (The exact query word is not mentioned in the previous work we are comparing against, so we chose a worst-case scenario for our experiments.) Our scheme performed better than [18] for all data sizes. Their scheme needs 17 ms, 34 ms and 53 ms for 4MB, 11MB and 16MB subsets of the Enron dataset respectively. Our scheme consumed 5 ms, 11 ms and 25 ms for 4MB, 8MB and 16MB subsets of the Enron dataset respectively. The search time grows proportionately to the size of the response. Figure 13 shows the search performance on the document dataset.

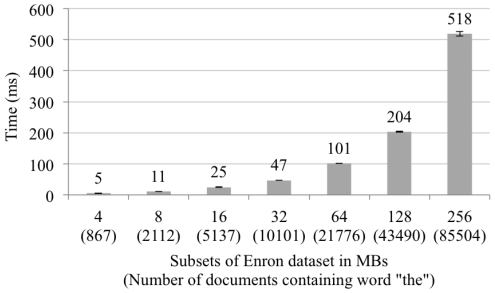

Fig. 12: Search performance on the email dataset with 99% confidence intervals

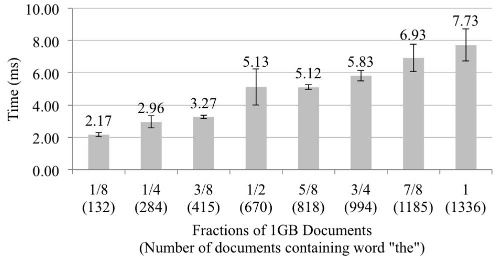

Fig. 13: Search performance on the document dataset with 99% confidence intervals

Note that our scheme uses a lazy deletion strategy to handle removals. This lazy delete mechanism allows us to obtain vastly improved security guarantees by limiting the information leaked to the server (only for files uploaded during initial index generation). One might ask if this leads to any efficiency degradation during subsequent searches, since the actual updates to the index take place when a keyword that was contained in a deleted document is searched for later.

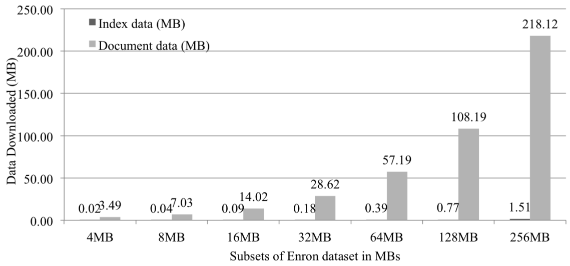

Fig. 14: Communication needed for searching on the email dataset. The graph shows the size of the retrieved documents themselves alongside the extra communication incurred by our scheme.

As it turns out (and as was experimentally confirmed), the overhead for searches does not significantly vary depending on whether the search operation involved a lazy deletion or not. This is because all search operations use the update mechanism of the underlying Blind-Storage scheme and the clear storage scheme. The efficiency of the update mechanism itself does not depend significantly on whether the file was modified or not. Indeed, in the case of Blind-Storage updates, it is important for the security that it must not be revealed to the server if a lazy deletion was involved or not.[16]

Communication costs. As our scheme does not involve any server-side computation, we download slightly more data compared to [18]. But as shown in Figure 14, for the email dataset, the communication overhead is negligible compared to the size of the documents retrieved. The document dataset is much richer and contains much fewer keywords compared to the email dataset of the same size (1GB of documents in our dataset contains only 70MB of text), and therefore the overhead would be even lower for it.

4) Add: As opposed to [18] and other prior work, performance of our *add* operation does not depend upon the amount of data (i.e. the number of file-keyword pairs) already present in the searchable encryption system. Figure 15 shows the performance of addition of files of specified size when 256MB of data was initially indexed into the system.

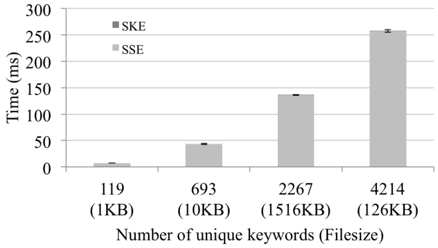

Fig. 15: Add performance on email dataset with 99% confidence intervals. SKE costs are non-zero but very small.

Communication costs. We only need, on average, to download three blocks and upload two blocks per unique keyword in the document that is being added. (If the server supports an

[16] We remark that our security model does not consider timing attacks. Depending on the implementation, we do not rule out a small dependence between the time taken and the extent of lazy delete computations involved. A serious implementation should take this into account. Since our SSE scheme is a relatively thin wrapper around the Blind-Storage mechanism, timing attacks can be effectively mitigated with relative ease.

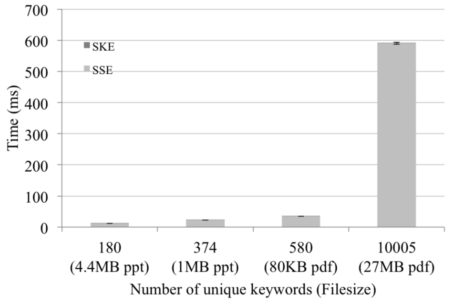

Fig. 16: Add performance on document dataset with 99% confidence intervals: SKE costs are non-zero but very small.

append operation that allows to append data to existing files on the server, we do not need to download any data during Add.)

5) Remove: The communication and computation cost of removing a document is virtually negligible, since it uses a lazy deletion strategy. Removal of a document in our scheme only requires the client to send a command to the server to delete the document from its file-system, and does not need any update to the searchable encryption index.

C. Summary

Evaluation of our scheme shows that it is more efficient, scalable and practical than prior schemes. Index generation in our scheme is more than *20 times* faster than that of [18]. Search operations are 2-3 times faster, in our experiments. Further, unlike [18], our addition and removal times are independent of the total number of file-keyword pairs, and is much more scalable. Removal in our scheme has virtually zero cost. We stress that several possible optimizations have not been implemented in this prototype.

VIII. Conclusion

In this work, we introduced a new cryptographic construct called Blind Storage, and implemented it using a novel, yet light-weight protocol ScatterStore. We also showed how a dynamic SSE scheme can be constructed using Blind Storage, in a relatively simple manner. The resulting scheme is more computationally efficient, require simpler infrastructure, and is more secure than the existing schemes.

Important future directions include making the scheme secure against *actively corrupt* servers, and allowing secure searches involving multiple keywords. The core idea of using pseudorandom subsets in ScatterStore is amenable to these extensions, as is being explored in on going work.

Acknowledgment

We thank Igors Svecs for collaboration in the early stages of the project. We are grateful to Seny Kamara for helping us with evaluation datasets and Elaine Shi for useful discussions. We also thank Ravinder Shankesi for help in debugging the code and Fahad Ullah for help in collecting documents for the document dataset.

This work was supported by NSF 07-47027, NSF 12-28856, NSF CNS 09-64392 (EBAM) and HHS 90TR0003-01 (SHARPS). The views expressed are those of the authors only.

References

[1] "Crypto++," http://www.cryptopp.com/.

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

[3] P. Brudenall, B. Treacy, and P. Castle, "Outsourcing to the cloud: data security and privacy risks," *Financier Worldwide and Hunton & Williams*, 2010.

[4] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," Electronic Colloquium on Computational Complexity (ECCC) TR01-016, 2001, previous version "A unified framework for analyzing security of protocols" available at the ECCC archive TR01-016. Extended abstract in FOCS 2001.

[5] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Roşu, and M. Steiner, "Dynamic searchable encryption in very large databases: Data structures and implementation," 2014.

[6] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *CRYPTO*, 2013.

[7] Y.-C. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data." in *ACNS*, 2005, pp. 442–455.

[8] M. Chase and S. Kamara, "Structured encryption and controlled disclosure," in *ASIACRYPT*, 2010, pp. 577–594.

[9] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, no. 5, pp. 895–934, 2011.

[10] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *CCS*, 2006, pp. 79–88.

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

[12] O. Goldreich, *Foundations of Cryptography: Basic Applications*. Cambridge University Press, 2004.

[13] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[14] P. Golle, J. Staddon, and B. R. Waters, "Secure conjunctive keyword search over encrypted data." in *ACNS*, 2004, pp. 31–45.

[15] W. Jansen and T. Grance, "Guidelines on security and privacy in public cloud computing," *NIST special publication*, pp. 800–144, 2011.

[16] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Outsourced symmetric private information retrieval," in *CCS*. ACM, 2013, pp. 875–888.

[17] S. Kamara and C. Papamanthou, "Parallel and dynamic searchable symmetric encryption," in *Financial Cryptography and Data Security, FC (2013)*, 2013.

[18] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *CCS*, 2012, pp. 965–976.

[19] K. Kurosawa and Y. Ohtaki, "UC-secure searchable symmetric encryption," in *Financial Cryptography and Data Security (FC)*, 2012.

[20] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *STOC*, 1990, pp. 514–523.

[21] S. Paquette, P. T. Jaeger, and S. C. Wilson, "Identifying the security risks associated with governmental use of cloud computing," *Government Information Quarterly*, vol. 27, no. 3, pp. 245 – 253, 2010.

[22] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *CRYPTO*, 2010, pp. 502–519.

[23] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data." in *IEEE S&P*, 2000, pp. 44–55.

[24] E. Stefanov, C. Papamanthou, and E. Shi, "Practical dynamic searchable encryption with small leakage," 2014.

[25] E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage," in *IEEE S&P*, 2013, pp. 253–267.

[26] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," in *NDSS*, 2012.

[27] P. van Liesdonk, S. Sedghi, J. Doumen, P. H. Hartel, and W. Jonker, "Computationally efficient searchable symmetric encryption," in *Workshop on Secure Data Management (SDM)*, 2010, pp. 87–100.