CS2: A Searchable Cryptographic Cloud Storage System

Seny Kamara
Microsoft Research
senyk@microsoft.com

Charalampos Papamanthou
UC Berkeley
cpap@cs.berkeley.edu *

Tom Roeder
Microsoft Research
throeder@microsoft.com

Abstract

Cloud storage provides a highly available, easily accessible and inexpensive remote data repository to clients who cannot afford to maintain their own storage infrastructure. While many applications of cloud storage require security guarantees against the cloud provider (e.g., storage of high-impact business data or medical records), most services cannot guarantee that the *provider* will not see or modify client data. This is largely because the current approaches for providing security (e.g., encryption and digital signatures) diminish the utility and/or performance of cloud storage.

This paper presents CS2, a cryptographic cloud storage system that guarantees confidentiality, integrity and verifiability without sacrificing utility. In particular, while CS2 provides security against the cloud provider, clients are still able not only to efficiently access their data through a *search* interface but also to *add* and *delete* files securely.

The CS2 system is based on new highly-efficient and provably-secure cryptographic primitives and protocols. In particular, we (1) construct the *first* searchable symmetric encryption scheme that is adaptively secure, dynamic and achieves sub-linear search time; (2) introduce and construct search authenticators (which allow a client to efficiently verify the correctness of search operations); and (3) design an efficient and dynamic proof of data possession scheme. Based in part on our new constructions, we propose two cryptographic protocols for cloud storage which we prove secure in the ideal/real-world paradigm. The first protocol implements standard keyword search. Our second protocol implements what we refer to as *assisted* keyword search, where a user performs a keyword search, sees a summary of the results and asks for a subset of these results.

Experimental results from an implementation of CS2 over both simulated and real-world data sets demonstrate that all operations achieve practical performance.

1 Introduction

Cloud storage promises high data availability, easy access to data, and reduced infrastructure costs by storing data with remote third-party providers. But availability is often not enough, as clients need guarantees about confidentiality and integrity for many kinds of data—guarantees that current cloud storage services cannot provide without prohibitive costs in computation and bandwidth. For example, confidentiality and integrity are essential for high-business impact enterprise data, secret government documents, and medical records. In this paper, we present CS2, a cryptographic cloud storage system that provides confidentiality, integrity and verifiability properties. We also present a prototype implementation that demonstrates the feasibility of CS2 in practice.

The need for cloud storage is increasing. According to studies conducted by the International Data Corporation [28, 27], the total amount of digital data generated by consumers and enterprises will grow next year to 1.2 zettabytes, i.e., 1.2 million petabytes. The increasing scale of stored data makes it

*Work done as an intern at Microsoft Research.

harder to store, manage, and analyze. Cloud storage services, such as Amazon S3 or Microsoft Azure Storage, offer several advantages to clients including elasticity (the ability to scale storage rapidly), universal access (the ability to access data from anywhere) and reliability. However, all of these services suffer from security and privacy concerns which have motivated research on secure cloud storage systems [31, 48, 52, 12, 33], namely the development of storage systems that can be layered on top of any key-based object store to provide various security guarantees to clients.

Confidentiality and integrity for cloud storage systems can be framed as follows: (1) confidentiality requires that the cloud provider not learn any information about the client's data and (2) integrity requires that the client detect any modification of its data by the provider. The standard approach to achieving these two properties in storage systems (cloud-based or not) is to encrypt and sign data at the client using symmetric encryption and digital signatures. Storage systems based on this approach are typically referred to as *cryptographic file systems* and provide end-to-end security in the sense that data is protected as soon as it leaves the client's possession. While cryptographic file systems provide strong security guarantees, they bring a high cost in terms of functionality, hence are inadequate for modern storage systems which, as argued above, must handle data at very large scales.

Two major limitations reduce the utility of cryptographic file systems. The first limitation is that data today is mostly accessed through search as opposed to file system navigation. The need for search in storage systems has long been recognized and has motivated a great deal of work on *semantic* file systems [29, 34, 40, 49] which eschew the traditional hierarchical file/folder organization for purely search-based access. The increasing importance of search has also led to the emergence of applications that index data for a user (or an enterprise) to allow for fast search. Search applications, such as Apple Spotlight, Google Dekstop and Windows Desktop Search, have quickly become indispensable, as search is now the most practical way of retrieving information.

Given the importance of search, we believe that any cloud storage system (secure or not) should be semantic, i.e., should provide the client search-based access to its data. There are two natural approaches to implement search over data encrypted with a traditional symmetric encryption scheme. The first is to store an index of the data locally and, for each search operation, query the index and use the results to retrieve the appropriate encrypted files from the cloud. The second approach avoids using local storage for indexes: instead, the index is stored in the cloud after being encrypted and signed. Then, for each search, the index is retrieved and queried locally before the encrypted files are fetched. As index sizes are normally non-trivial fractions of the datasets they index, it is clear that neither approach scales.

The second limitation is that cryptographic file systems guarantee integrity only for data retrieved by a user. This is an inherent limitation of integrity mechanisms such as message authentication codes and signatures because they require knowledge of all the data for verification. While this kind of *local* integrity guarantee may be sufficient for small datasets since a user can retrieve all the data, it is not appropriate for large-scale data, where substantial portions may be accessed only rarely. In this context, a better guarantee would be *global* integrity: a client should be able to verify the integrity of all its data—regardless of whether the data has been retrieved.

CS2 addresses these issues with practical protocols and a semantic data interface. The main goals of CS2 are to provide the client with (1) *search*: access to its data based on keywords; (2) *confidentiality*: assurance that the cloud provider learns no information about the data; (3) *global integrity*: assurance that no data has been modified; and (4) *verifiability*: the ability for clients to check that operations were performed correctly by the provider. For instance, the client should be able to determine efficiently if the set of files returned for a search query is correct. Finally, CS2 supports *efficient* dynamic updates of data: namely, the (remotely) stored data does not need to be recovered and re-processed whenever an update (e.g., addition or deletion of a file) occurs.

Our approach At a very high level, we achieve these properties by designing new cryptographic primitives and new cryptographic protocols that are better suited to the cloud storage setting. These include symmetric searchable encryption (SSE), search authenticators (SA) and proofs of storage (PoS). SSE allows a client to encrypt data in such a way that it can later generate *search tokens* for a storage provider. Given such a token, the provider can search over the encrypted data and return the appropriate encrypted files without learning anything about the data or the search query. Using SSE, CS2 can provide search with confidentiality in a cloud setting in a scalable way (without having to store an index locally). However, SSE does not provide a way for a client to determine whether the server returned the correct files in response to a search query. Of course, the client could decrypt the files and check that they indeed contain the queried term, but the server could omit other files that also contain the term. We address this by introducing a new cryptographic primitive: search authenticators, which allow the cloud provider to *prove* to the client that it returned all and only the correct files. Global integrity is achieved using a PoS, which is a protocol that allows a client to verify the integrity of its data without having access to it.

We note that for our purposes the SSE, SA and PoS schemes that underlie our protocols must be *dynamic*, i.e., must allow for the secure addition and deletion of files. While dynamic SSE [15, 30] and PoS [4] schemes are known, they are not efficient enough for our purposes. In addition, the SSE schemes of [15, 30] do not satisfy the level of security we seek (i.e., security against *adaptive* chosen-keyword attacks). To address this, in addition to introducing new protocols, we also construct new highly-efficient *dynamic* and *provably-secure* SSE, SA and PoS schemes.

At the time of this writing CS2 is a personal storage system, since it does not allow clients to share files. The design of CS2, however, is flexible enough that it could employ one of several sharing mechanisms. We discuss this further in the full version [38], describing several approaches for sharing.

Our contributions We make several contributions in this work which include (1) the design of new provably-secure cryptographic primitives and protocols; and (2) the implementation and evaluation of a system based on our new cryptographic tools. More precisely, our cryptographic contributions include:

1. The first formal security definition for *dynamic* SSE which captures the strongest notion of security for SSE, namely *adaptive* security against chosen-keyword attacks (CKA2) [18].

2. The first SSE scheme that is dynamic, CKA2-secure and achieves sub-linear search (in fact, our construction achieves optimal search time). Unlike previously known schemes [53, 30, 15, 18, 16], our construction is secure in the random oracle model.

3. The introduction and formalization of (dynamic) search authenticators which are verifiable computation schemes for search. We observe that the composition of search authenticators and SSE is subtle and address this by introducing and formalizing the notion of a *hiding* search authenticator. Two notable properties of our search authenticator construction are that it is dynamic and that proofs can be constructed in sub-linear time (in the number of files). Due to space limitations, we defer the description of our dynamic SA construction to the full version of this work [38].

4. A formal security definition for secure cloud storage in the ideal/real-world paradigm [14], which has several advantages over game-based definitions including better composition properties.

5. Two protocols for secure cloud storage which we prove secure according to our ideal/real-world definition. Our protocols make black-box use of dynamic SSE, SA and PoS schemes and relies on collision-resistant hash functions. The first protocol securely implements (standard) search where

a user requests all files with a particular keyword. Our second protocol securely implements what we refer to as *assisted* search, where a user performs a keyword search, sees summaries of the results and asks for a subset of these result. It includes optimizations that make it particularly efficient for applications that interface with a human, e.g., a cloud-based email system. Again, due to space limitations we only present here our protocol for assisted search and refer the reader to [38] for a detailed description of our protocol for standard search.

Our second set of contributions consist of our implementation and evaluation of a cryptographic cloud storage system based on our new cryptographic primitives and protocols. More precisely, these include:

6. The first implementation and evaluation of a sub-linear SSE scheme, in particular, the first implementation of an *inverted index*-based construction along the lines of [18]. Our implementation shows that this type of SSE scheme can be extremely efficient. We also implement and evaluate our search authenticator and dynamic PDP schemes.

7. A synthetic Zipf [55] model of file collections that we use to evaluate our cloud storage service. We validate this model in micro-benchmarks against real-world data sets.

8. A performance evaluation of SSE, search authenticators, and PDP that shows the incremental cost of adding each security property (i.e., confidentiality, verifiability and integrity); this demonstrates the relative costs of increasing levels of security.

Due to space limitations we focus on describing the CS2 system and reporting on our experimental results in favor of describing our cryptographic constructions. We refer the reader to the full version of this work [38] for detailed descriptions, formal security definitions and proofs of security.

2 Related Work

Due to space limitations we only provide an overview of work that is directly related to our own. For an extended bibliographic study, see the full version of this work [38].

Cryptographic file systems provide "end-to-end security", i.e., they guarantee confidentiality and integrity even against an untrusted storage service. There has been a lot of work on cryptographic file systems and we refer the reader to the survey by Kher and Kim [39] for further details.

Plutus [37] is a cryptographic file system that provides sharing on a per-file basis, but does not achieve secure key revocation [26]. SiRiUS [31] is a cryptographic cloud storage system[1] that allows for sharing on a per-file basis and a weak form of freshness, i.e., the metadata of a file cannot be replaced with an older version of itself (though the file itself can be). CloudProof [48] is a cryptographic cloud storage system that provides read/write sharing. It achieves confidentiality, integrity, fork-consistency [42] and freshness (for both data and meta-data). Roughly speaking, fork-consistency guarantees that an untrusted storage service cannot provide different versions of the data to different clients without being detected. In addition to these properties, CloudProof allows the parties in the system to prove to a third party that any of these properties have been violated. As discussed in the Introduction, end-to-end security is typically achieved by encrypting and signing data at the client using traditional cryptographic techniques like symmetric encryption and digital signatures or message authentication codes. As such, unlike CS2, none of the previously proposed storage systems achieve scalable search or global integrity.

[1] While SiRiUS was implemented on top of NFS, it was designed so that it could be layered on top of any storage system.

The problem of searching on symmetrically encrypted data can be solved in its full generality using oblivious RAMs [32]. Unfortunately, these methods are not very efficient, which is to be expected given that they provide a very strong notion of security (e.g., not even the access pattern is revealed).

Searchable encryption was considered explicitly by Song, Wagner and Perrig in [53], where they give a non-interactive solution that achieves search time that is linear in the length of the file collection. In [30], Goh introduced formal security definitions for symmetric searchable encryption and proposed a construction based on Bloom filters [8] that requires $O(n)$ search time on the server (where n is the number of files) and results in false positives. Goh's index-based approach to constructing SSE schemes was adopted by Chang and Mitzenmacher [15] who proposed alternative security definitions and a new construction also with $O(n)$ search time but without false positives.

In [18], Curtmola, Garay, Kamara and Ostrovsky gave the first constructions (SSE-1 and SSE-2) to achieve sub-linear (and in fact optimal) search time. Like previous work [30, 15], SSE-1 was shown secure against chosen-keyword attacks (CKA1). However, it was also pointed out in that work that CKA1-security does not suffice for practical use. To address this, the stronger notion of security against *adaptive* chosen-keyword attacks (CKA2) was proposed and a sub-linear CKA2-secure SSE scheme (SSE-2) was proposed. Unfortunately, the latter is not practical as the constants in the asymptotic search time are high. A similar scheme was described by Chase and Kamara [16] but its space complexity is poor. In addition, none of these schemes are "explicitly dynamic", i.e., to handle dynamic data one must use general dynamization techniques that are relatively inefficient. Our construction, on the other hand, is CKA2-secure, achieves sub-linear (and optimal) search with small constants, handles dynamic data explicitly (i.e., without dynamization) and has good space complexity. We note that, unlike the schemes of [18] and [16], our construction is in the random oracle model. Starting with the work of Boneh, Di Crescenzo, Ostrovsky and Persiano [9], searchable encryption has also been considered in the public-key setting [9, 54, 1, 13, 10, 11, 51].

Proofs of storage are interactive protocols that allow a client to verify the integrity of remotely stored data. There are two variants of PoS: (1) proofs of data possession (PDP), introduced by Ateniese et al. [2]; and (2) proofs of retrievability (PoR), introduced by Juels and Kaliski [36]. Earlier work by Naor and Rothblum [45] is closely related but considers a weaker adversarial model. Roughly speaking, a PDP guarantees that tampering will be detected if it is above a certain threshold. A PoR, however, can provide the additional guarantee that the data is recoverable if the tampering is below a certain threshold. Extensions and improvements to both PDPs and PoRs were given in [50, 4, 20, 12, 3, 22]. The CS2 system only provides the weaker PDP guarantee. The reason is that PoRs require the use of an erasure code, which prevents us from handling dynamic data. One approach to constructing PoRs and PDPs is from a homomorphic linear authenticator (HLA) [2]. In this work, we construct a dynamic PDP scheme from the underlying HLA used in Shacham and Waters' privately-verifiable PoR [50].

Relevant to this work are also authenticated data structures [44, 41, 47, 46, 33] which allow the verification of data structure operations. Our dynamic search authenticator construction (described in the full version [38]) can be seen as an authenticated data structure for the multi-map abstract data type.

3 Overview

Two parties interact in our system. The first is a cloud provider that provides access to a key-based object store. The second is a client that stores its data with the provider. The former is untrusted and may act maliciously, for example, by trying to recover information about the client's data and queries, by tampering with the client's data or by answering search queries incorrectly.

The client's data can be viewed as a sequence of n files $\mathbf{f} = (f_1, \ldots, f_n)$; we assume that each file

has a unique identifier, which we refer to as $id(f_i)$. The client's data is dynamic, so at any time it may add or remove a file. We further assume the client has access to an indexing program Index that takes as input a sequence of files $\mathbf{f}$ and outputs an inverted index δ. We note that the files do not have to be text files but can be any type of data as long as CS2 is provided with an Index operation that labels and indexes the files. Given a keyword w we denote by $\mathbf{f}_w$ the set of files in $\mathbf{f}$ that contain w. If $\mathbf{c} = (c_1, \ldots, c_n)$ is a set of encryptions of the files in $\mathbf{f}$, then $\mathbf{c}_w$ refers to the ciphertexts that are encryptions of the files in $\mathbf{f}_w$.

CS2 provides six operations to clients: SETUP, STORE, SEARCH, CHECK, ADD and DELETE. SETUP sets up the system and generates keying material for the client. The STORE operation is used by the client to store a set of files $\mathbf{f} = (f_1, \ldots, f_n)$ with the provider. STORE indexes the files and processes them before sending them to the provider. The SEARCH operation is used by the client to search for the files labeled with a particular keyword. Note that using SEARCH, a user can retrieve files by their unique identifiers since CS2 labels each file with its unique identifier. The CHECK operation allows the client to verify the integrity of its data. Finally, ADD and DELETE are used to add and delete files.

Most of these operations are *keyed* and *stateful* and the client is responsible for keeping its key material and state secret. More precisely, CS2 provides the following API to the client:

- SETUP(k): takes as input a security parameter k and returns the user's secret key K.
- STORE($K, \mathbf{f}$): takes as input the user's secret key K and a sequence of files $\mathbf{f} = (f_1, \ldots, f_n)$. It indexes the files using Index and processes them using various cryptographic primitives before sending them to the provider. It returns some state st to be kept secret from the provider.
- SEARCH(K, w, st): takes as input the user's secret key K, a keyword w and the state st. With the standard search option, the function interacts with the provider and returns either a sequence of files $\mathbf{f}_w \subseteq \mathbf{f}$ or $\perp$ if the provider did not send back the correct (encrypted) files. With the assisted search option, the operation returns summaries for the files $\mathbf{f}_w$ and waits for the user to choose a subset $\mathbf{f}'_w \subset \mathbf{f}_w$. The files $\mathbf{f}'_w$ are then downloaded from the provider and returned to the user.
- CHECK(K, st): takes as input the user's secret key K and the state information st. After interacting with the provider, it returns true if the provider tampered with the data and false otherwise.
- ADD(K, st, f): takes as input the user's secret key K, the state information st, and a file f. After interacting with the provider, it returns an updated state st'.
- DELETE(K, st, id): takes as input the user's secret key K, the state information st, and a file identifier id. After interacting with the provider, it returns an updated state st'.

In our implementation, the user's secret key is 416 bytes and the state is 48 bytes along with 4 bytes per modified block.

4 The CS2 Protocol for Assisted Search

In this section we describe how CS2 implements the operations it exposes to clients and give a high-level description of our SSE construction. First, however, we describe the underlying cryptographic primitives we make use of (formal definitions are provided in the full version of this work). For each operation, we give an identifier (in parentheses) that will be used to refer to this operation in the evaluation.

4.1 Building Blocks

Symmetric searchable encryption. Searchable encryption allows a client to encrypt data in such a way that it can later generate *search tokens* for a storage server. Given a search token, the server can search over the encrypted data and return the appropriate encrypted files. CS2 makes use of a particular type of (symmetric) searchable encryption scheme which we refer to as *index-based*. The encryption algorithm (SSE.Enc) of index-based searchable encryption takes as input an index δ and a sequence of n files $\mathbf{f} = (f_1, \ldots, f_n)$ and outputs an encrypted index γ and a sequence of n ciphertexts $\mathbf{c} = (c_1, \ldots, c_n)$. All known index-based constructions [30, 15, 18] can encrypt the files $\mathbf{f}$ using any symmetric encryption scheme, i.e., the file encryption does not depend on any unusual properties of the encryption scheme. The encrypted index γ and the ciphertexts $\mathbf{c}$ do not reveal any information about $\mathbf{f}$ other than the number of files n and their length[2], so they can be stored safely at an untrusted cloud provider. To search for a keyword w, the client generates a search token τ_w (SSE.SrchToken) and given τ_w, γ and $\mathbf{c}$, the provider can find the identifiers $\mathbf{I}_w$ of the files that contain w (SSE.Search). From these identifiers it can recover the appropriate ciphertexts $\mathbf{c}_w$. Notice that the provider learns some limited information about the client's query. In particular, it knows that whatever keyword the client is searching for is contained in whichever files resulted in the ciphertexts $\mathbf{c}_w$ (of course, it does not learn anything about the files, since they are encrypted). There are ways to hide even this information, most notably using the approach of Goldreich and Ostrovsky [32], but such an approach leads to inefficient schemes. A more serious limitation of known SSE constructions (including ours) is that the tokens they generate are deterministic, in the sense that the same token will always be generated for the same keyword. This means that searches leak statistical information about the user's search pattern. Currently, it is not known how to design efficient SSE schemes with probabilistic trapdoors. Recall that CS2 handles dynamic data so the underlying SSE scheme must handle addition and deletion of files. Both of these operations are handled using tokens. To add a file f, the client generates an add token τ (SSE.AddToken) and given τ and γ, the provider can update the encrypted index γ (SSE.Add). Similarly, to delete a file f, the client generates a delete token τ' (SSE.DelToken), which the provider uses to update γ (SSE.Del).

Proofs of data possession. A PDP is a protocol executed between the client and the provider that allows the client to verify the integrity of its files $\mathbf{f}$ without needing to have downloaded them. Similarly to SSE, there are several types of PDP, and CS2 makes use of a particular kind we refer to as *tag-based*. In a tag-based PDP, the client tags its files (PDP.Encode), which results in a sequence of tags $\mathbf{t} = (t_1, \ldots, t_n)$ and some state information st. The state information has to be kept secret by the client but it is very short (e.g., 32 bytes plus 4 bytes per modified block in our implementation). The files, together with the tags, are then sent to the provider. From this point on the client can verify the integrity of its files at any point by sending a challenge to the server (PDP.Chall). The latter uses the challenge, the files and the tags to generate a short proof π that it returns to the client (PDP.Prove). Finally, the client uses the state st to verify the proof (PDP.Vrfy). Notice that no matter how large the file collection is, two properties hold: (1) the total amount of information stored at the client is very small, i.e., a secret key and a small state; and (2) the total amount of information exchanged by the parties is also very small, i.e., a short challenge and a short proof. Also, the client can request an addition (PDP.Add) or a deletion (PDP.Del) of a file, which is achieved in our system by a special application of PDP.Encode. File addition or deletion causes the state information st to be updated. PDP schemes can be either privately or publicly verifiable. Privately verifiable PDP schemes only allow the client (i.e., the party that encoded the file) to verify integrity, while a publicly verifiable scheme allows anyone that possesses the client's public key to verify. In CS2, we only make use of privately-verifiable PDP schemes. Finally, to add a file f, the client generates a tag t and sends the

[2]Note that this information leakage can be mitigated by padding if desired.

new file together with its tag to the provider. This process results in a new state which the client stores (replacing the old one). To delete a file, it suffices for the client to send the identifier of the file and to update its state locally. Our dynamic PDP scheme (described in the full version) extends the (static) privately-verifiable HLA of Shacham and Waters based on pseudo-random functions [50].

Search authenticators. A dynamic search authenticator allows a server to prove to a client that it answered a search query correctly. The client, on input its files and index, begins by creating the authenticator α and state information st (DSA.Auth). The files, together with the authenticator are then sent to the provider. State st is locally stored by the client. When the provider answers a search query, it uses the authenticator and the files to generate a proof π that it returns to the client (DSA.Prove). The latter can then use its state st and the proof π to verify the returned files (DSA.Vrfy). To add or remove a file, the client sends an add token output by DSA.AddToken to the provider. Note that here we assume the client will always have a local copy of any file it wishes to remove.[3] The provider then updates the authenticator (DSA.Add or DSA.Del) and sends a receipt ρ back to the client who will update its state st. For our purposes, we need search authenticators that are not only dynamic but also *hiding* in the sense that the authentiactor α reveals no information about the files. In the full version, we show how to construct such authenticators based on Merkle hash trees, incremental hash functions [6] and pseudo-random functions.

4.2 The Protocol

We now describe our protocol for assisted search. A precise and more formal description as well as a description of our standard search protocol is provided in the full version of this work [38].

- SETUP is run by the client to setup the user's secret key K, which is composed of keys for the underlying cryptographic primitives described in the previous sections.
- STORE is run by the client to store a set of files $\mathbf{f} = (f_1, \ldots, f_n)$ at the provider. The client starts by running the indexing program Index on the files $\mathbf{f}$. This results in an index δ, which it encrypts together with $\mathbf{f}$ using the SSE scheme. This step yields an encrypted index γ and a set of encrypted files $\mathbf{c}$. It then proceeds to encode the ciphertexts $\mathbf{c}$ using the PDP; to authenticate δ and a sequence of headers $\mathbf{h} = (h_1, \ldots, h_n)$ with the DSA (i.e., the headers are used in place of the files). Here, each $h_i = (id(f_i), s_i, \mathrm{H}(c_i))$ where s_i is a (private-key) encryption of a summary of the file f_i and H is a collision-resistant hash function . These two operations result in a set of tags $\mathbf{t}$ (one for each ciphertext in $\mathbf{c}$), an authenticator α, and a state st. Finally, the client sends the provider the encrypted index γ, the encryptions of the files $\mathbf{c}$, the set of tags $\mathbf{t}$, the headers $\mathbf{h}$ and the authenticator α.
- SEARCH is a protocol between the client and the cloud provider to search for all files with keyword w. The client starts by generating an SSE search token τ for its keyword w. The token is sent to the provider who uses it to recover the identifiers $\mathbf{I}_w$ of the (encrypted) files that contain w. The provider then sends back the set $\{h_i \in \mathbf{h} : i \in \mathbf{I}_w\}$ and a DSA proof π_a. The client verifies the DSA proof and if valid, it uses its secret key to decrypt the summaries of the files and returns them to the user. The user then returns a subset $\mathbf{I}' \subset \mathbf{I}_w$ of these files which the client sends to the provider. The latter then sends back the ciphertexts $\{c'_i \in \mathbf{c} : i \in \mathbf{I}'\}$. For $i \in \mathbf{I}'$, the client verifies that $\mathrm{H}(c'_i)$ is equal to the appropriate hash in $\mathbf{h}$. If any of these verifications fail it outputs $\perp$, otherwise it decrypts the ciphertexts and returns the files.

[3] If this is not the case, the client can always retrieve the file by its unique identifier before generating the token.

- CHECK is a protocol between the client and the provider to verify whether the provider tampered with the client's files. The client sends a PDP challenge to the provider who responds with a PDP proof π_p computed using the ciphertexts $\mathbf{c}$ and the tags $\mathbf{t}$. The proof is then verified by the client and true is returned if verification goes through and false otherwise.
- ADD is a protocol between the client and the provider to add a file. First the client sends the file, its PDP tag and SSE and DSA add tokens to the provider. The provider returns a DSA receipt ρ to the client who uses it to update its DSA state.
- DELETE works analogously to ADD.

Security. The protocol is designed so that the CS2 system behaves (roughly) as a trusted storage service even when it is used with an untrusted cloud storage provider. More precisely, CS2 guarantees:

1. (confidentiality) the provider cannot learn any information about the client's files,
2. (integrity) the provider cannot tamper with the client's files without being detected,
3. (verifiability) the provider cannot add or omit files from the results of a SEARCH operation without being detected.

In the full version, we formalize the security properties achieved by CS2 (and informally discussed above) in the ideal/real-world paradigm which is traditionally used to evaluate the security of cryptographic protocols. This paradigm compares the real-world execution of a protocol with an ideal-world execution by a trusted party [14].

At a very high level, confidentiality is achieved by using SSE, which guarantees that client data and search terms remain hidden from the provider. Integrity is guaranteed through the use of PDP and verifiability is achieved with search authenticators. We stress, however, that this is only a rough intuition and that in practice the composition of cryptographic primitives can lead to subtle issues that can break security.

4.3 A Concrete SSE Construction

Our SSE scheme is an extension of the SSE-1 construction from [18, 19], which is based on the inverted index data structure. Though SSE-1 is practical (it is asymptotically optimal with small constants), it does have limitations that make it unsuitable for direct use in CS2: (1) it is only secure against non-adaptive chosen-keyword attacks (CKA1) which, intuitively, means that it can only provide security for clients that perform searches in a batch; and (2) it is not fully dynamic, i.e., it can only support dynamic operations using general and inefficient techniques.

Before discussing how we address these issues, we first recall the SSE-1 construction at a high level (for more details we refer the reader to [18, 19]). The scheme makes use of a private-key encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, two pseudo-random functions F and G, an array $\mathtt{A}_s$ we will refer to as the *search array* and a lookup table $\mathtt{T}_s$ we refer to as the *search table*.

The SSE-1 construction. To encrypt a collection of files $\mathbf{f}$, the scheme constructs a list $\mathtt{L}_w$ for all keywords $w \in W$. Each list $\mathtt{L}_w$ is composed of $\#\mathbf{f}_w$ nodes $(\mathtt{N}_1, \ldots, \mathtt{N}_{\#\mathbf{f}_w})$ that are stored at random locations in the search array $\mathtt{A}_s$. Node $\mathtt{N}_i$ is defined as $\mathtt{N}_i = \langle id, \mathsf{addr}(N_{i+1}) \rangle$, where id is the unique file identifier of a file that contains w and $\mathsf{addr}(\mathtt{N})$ denotes the location of node $\mathtt{N}$ in $\mathtt{A}_s$. To prevent the size of $\mathtt{A}_s$ from revealing statistical information about $\mathbf{f}$, it is recommended that $\mathtt{A}_s$ be of size at least $|\mathbf{c}|/8$ and that the unused cells be padded with random strings of appropriate length.

For each keyword w, a pointer to the head of $\mathtt{L}_w$ is then inserted into the search table $\mathtt{T}_s$ under search key $F_{K_1}(w)$, where K_1 is the key to the PRF F. Each list is then encrypted using SKE under a key generated as $G_{K_2}(w)$, where K_2 is the key to the PRF G.

The reason the nodes are stored at random locations in $\mathtt{A}$ is to hide the length of the lists which could reveal information about the frequency distribution of words in $\mathbf{f}$ [4]. It is important to note, however, that random storage by itself is not enough to hide the length of the lists since the encryption scheme used to encrypt the nodes could potentially reveal to an adversary which nodes are encrypted under the same key (standard notions of security for encryption such as CPA-security do not prevent this). To address this each node $\mathtt{N}_i$ is encrypted under a different key $K_{\mathtt{N}_i}$ which is then stored in node $\mathtt{N}_{i-1}$.

To search for a keyword w, it suffices for the client to send the values $F_{K_1}(w)$ and $G_{K_2}(w)$. The server can then use $F_{K_1}(w)$ with $\mathtt{T}_s$ to recover the pointer to the head of $\mathtt{L}_w$, and use $G_{K_2}(w)$ to decrypt the list and recover the identifiers of the files that contain w. As long as $\mathtt{T}$ supports $O(1)$ lookups (which can be achieved using a hash table), the total search time for the server is linear in $\#\mathbf{f}_w$, which is optimal.

Making SSE-1 dynamic. As mentioned above, the limitations of SSE-1 are twofold: (1) it is only CKA1-secure and (2) it is not explicitly dynamic. As observed in [16], the first limitation can be addressed relatively easily by requiring that SKE be non-committing (in fact the CKA2-secure SSE construction proposed in that work uses a simple PRF-based non-committing encryption scheme).

The second limitation, however, is less straightforward to overcome. The difficulty is that the addition, deletion or modification of a file requires one to add, delete or modify nodes in the encrypted lists stored in $\mathtt{A}_s$. This is difficult for the server to do since: (1) upon deletion of a file f, it does not know where (in $\mathtt{A}$) the nodes corresponding to f are stored; (2) upon insertion or deletion of a node from a list, it cannot modify the pointer of the previous node since it is encrypted; and (3) upon addition of a node, it does not know which locations in $\mathtt{A}_s$ are free.

At a high level, we address these limitations by constructing the first *dynamic* SSE scheme. The proof of security for the construction (described in Appendix A) can be found in the full version [38].

1. (file deletion) we add an extra (encrypted) data structure $\mathtt{A}_d$ called the *deletion array* that the server can query (with a token provided by the client) to directly recover pointers to the nodes that correspond to the file being deleted, wtihout having to traverse the list of nodes that are indexed under the same keyword. More precisely, the deletion array stores for each file f a list $\mathtt{L}_f$ of nodes that point to the nodes in $\mathtt{A}_s$ that should be deleted if file f is ever removed. So every node in the search array has a corresponding node in the deletion array and every node in the deletion array points to a node in the search array. Throughout, we will refer to such nodes as *duals* and write $\mathtt{N}^\star$ to refer to the dual of a node $\mathtt{N}$.

2. (pointer modification) we encrypt the pointers stored in a node with a homomorphic encryption scheme. By providing the server with an encryption of an appropriate value, it can then modify the pointer without ever having to decrypt the node.

3. (memory management) to keep track of which locations in $\mathtt{A}_s$ are free we add a *free list* that the server can query (given an appropriate token from the client) to get a pointer to a free slot in $\mathtt{A}_s$.

[4] Of course, the lists could be stored at fixed locations if they were all padded to the same length but this would waste storage.

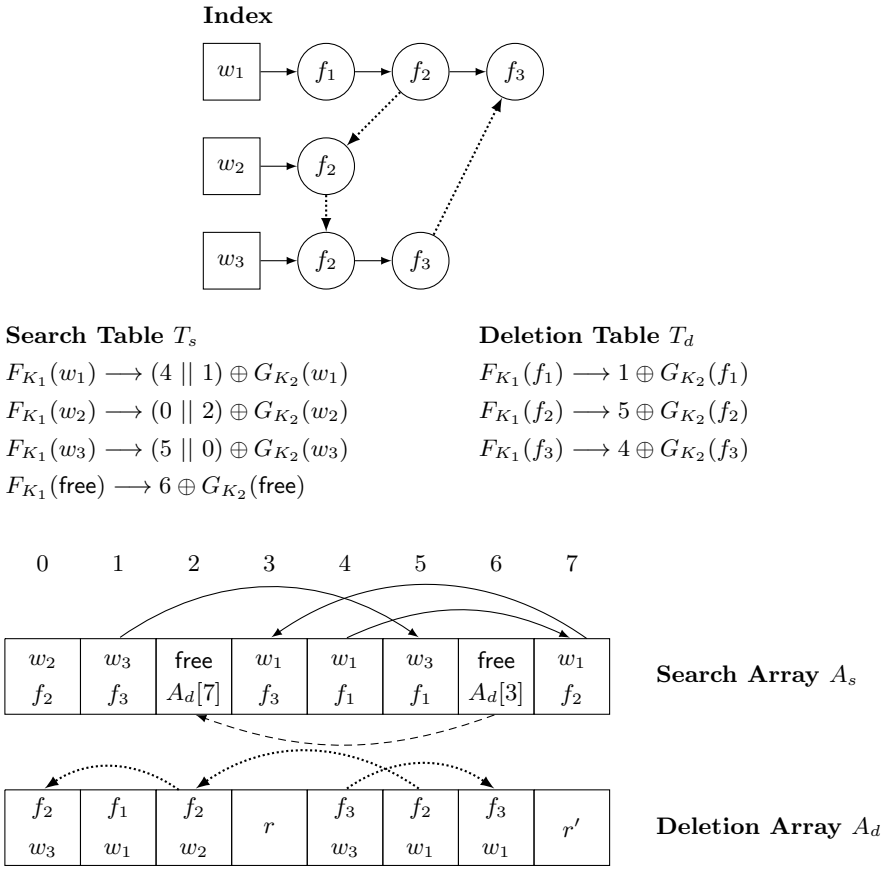

Figure 1: A small example of a dynamic encrypted index.

The dynamic SSE data structures. Figure 1 describes the dynamic SSE data structures over a toy index containing 3 files and 3 unique words. The first part of the diagram shows the connections between the elements of the index. The second part of the diagram shows the generated SSE index and represents (encrypted) pointers between array elements as arrows.

5 Implementation

To demonstrate the feasibility of our algorithms, we implemented CS2 in C++ over the Microsoft Cryptography API: Next Generation (CNG) [17] and the Microsoft Bignum library (the core cryptographic library in Microsoft Windows). Our implementation uses the algorithms described in §4. The cryptographic primitives for our protocols were realized as follows.

- Encryption is the CNG implementation of 128-bit AES-CBC [25].
- The hash function is the CNG implementation of SHA-256 [23], and is used as a random oracle.
- SSE employs a two-argument random oracle, which is implemented using HMAC-SHA256 from CNG (this employs the HMAC construction first described by Bellare, Canetti, and Krawczyk [5]). The first parameter passed to the random oracle is used as a key to the HMAC, and the second parameter is used as input to the HMAC (in the SSE construction, the first parameter passed to the two-argument random oracle is always 16 bytes of randomness).
- Prime fields and large integer arithmetic are implemented in Bignum.

A system that implements CS2 performs two classes of time-intensive operations: cryptographic computations (e.g., SSE, DSA, and PDP) and systems actions (e.g., network transmission and filesystem access). To separate the costs of cryptography from the systems costs (which will vary between implementations), we built a test framework that performs cryptographic computations on a set of files but does not transfer these files across a network or incur the costs of storing and retrieving tags, indexes, or verification information from disk; all operations are performed in memory. We also ignore the cost of producing a plain-text index for the files, since the choice and implementation of an indexing algoritm is orthogonal to CS2. Note, however, that given the performance numbers we present in §6, the cost of transferring files across the network is likely to dominate in the total execution time for CS2.

Dynamic SSE. We implemented the file-based dynamic SSE scheme described in §4 and Appendix A (see Figure 1 for a toy example). CS2 also provides search and update authenticators for SSE operations as described in the same section; the computation and verification of these authenticators was integrated directly into the SSE implementation.

Authenticators. Our authenticator implementation depends on two components: Merkle hash trees [43] and incremental hash functions [7]. Our incremental hash functions employ the multiplicative hash MuHash described by Bellare and Micciancio [7]; the incremental hash of a set of file identifiers for a given word is computed as follows: the client keeps a PRF key and computes the PRF of the encryption of each file identifier (of size about 1kB), along with a 16-byte IV and the hash of the encryption of the file. The output of the PRF is hashed (using SHA-256), and the hash values are treated as elements of a 2048-bit DSA group [24]; these values are multiplied to get the output of the incremental hash function. Adding or removing file identifiers from the incremental hash value involves computing the PRF and hash, then multiplying or dividing, respectively.

6 Experiments

Cryptographic operations in CS2 require widely varying amounts of time to execute. So, to evaluate CS2, we performed micro-benchmarks and full performance tests on the system and broke each test out into its component algorithms. The micro-benchmarks are used to explain the performance of the full system and to justify the selection of parameters for the algorithms. The full tests allow us to determine the incremental cost of adding layers of security to a cloud storage system.

These experiments were performed on an Intel Xeon CPU 2.26 GHz (L5520) running Windows Server 2008 R2. All experiments ran single-threaded on the processors. Each data point presented in the experiments is the mean of 10 executions, and the error bars provide the sample standard deviation.

The unit of measurement in all of the SSE experiments is the *file/word pair*: for a given file f, the set of file/word pairs is comprised of all unique pairs (f, w) such that w is a word associated with f in the index. The set of all such tuples across all files in a file collection is exactly the set of entries in an index for this collection. And file f is associated with a *file structure* f' that is a fixed-length representation of f, containing path, description, $id(f)$, and other information useful in searches.[5]

6.1 Standard Search

The original search construction of Curtmola et al. defines searchable symmetric encryption as a single-round operation; a client sends a search query to a server, which responds with the files. In assisted

[5] In this section, we often abuse notation for the sake of brevity and call a file structure a file.

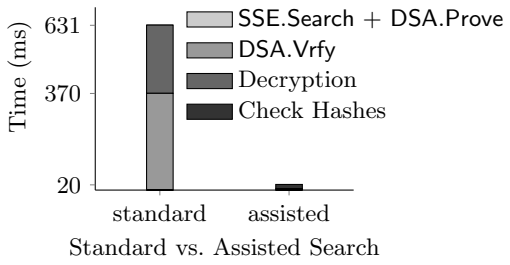

Figure 2: Execution time for standard and assisted SEARCH.

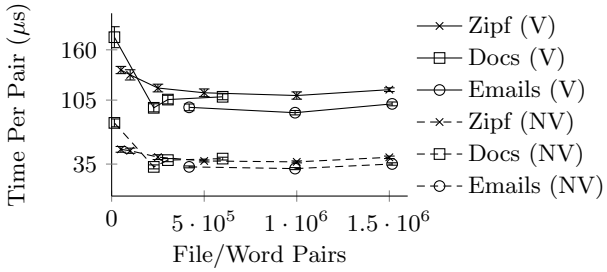

Figure 3: Execution time for SSE.Enc, with (V) and without (NV) DSA.Auth

search, however, a client receives only encrypted file structures, along with the hash of the encrypted file, from the server in response to its query. Then, the client decrypts the file structures, chooses a subset of files, and requests these encrypted files from the server (the client can check that it received the correct file by computing the hash of the file it receives and comparing it with the hash it received in response to its search query).

The cost of assisted search can be significantly lower than standard search in many practical circumstances, since clients normally do not require all documents that match a given search query; the final result of most searches is a small number of documents. Standard search requires time linear in the size of the returned documents, while assisted search requires time linear in only the number of the returned documents.

To analyze the cost of standard search, we present the results of a search query on a 500MB media repository; the query returns 40MB of encrypted documents, so the network transfer time on a 10Mbit connection would be about 32 seconds. By contrast, the two-round version returns 70 kB of encrypted file structures, which would require about 55 ms to transfer. The following graph presents the relative differences in search time between standard and assisted search.

In Figure 2, the time needed to check the search authenticators and decrypt the files depends linearly on the number of bytes returned, so it is much longer in the standard search version: 631 ms as compared to about 20 ms. Assisted search must additionally hash the encrypted files and compare them against the hashes it receives, but it only performs this hash on a fixed number of files. In this graph, we assume that the user requests 3 files out of the search. This demonstrates the potential relative benefit of assisted search over standard search.

6.2 Micro-Benchmarks

SSE and DSA. To determine the performance of SSE, we generated synthetic indexes and executed search and update operations on them. For searches, we chose the word that was present in the most files. And we deleted and added back in a file with the largest number of unique words in the index.

We generated our synthetic indexes from a pair of Zipf distributions [55] with parameter $\alpha = 1.1$; one distribution contained randomly-generated file structures, and the other contained words (the words in our case were simply numbers represented as strings: "0", "1", "2", etc.). The synthetic file collection was generated as follows. First, the test code drew a file f from the Zipf file distribution (our sampling employed the algorithm ZRI from Hörman and Derflinger [35]). Second, the test code drew words from the word distribution until it found a word that was not in f It then added this word to the index information for f and drew another file to repeat the process. This process corresponds to writing a set of files with Zipf-distributed sizes and containing Zipf-distributed words such that the file collection as a whole contains a given number of file/word pairs.

We chose two sets of real-world data to validate the synthetic data. The first set was selected

from the Enron emails [21]; we extracted a subset of emails and used decreasing subsets of this original subset as file collections with different numbers of file/word pairs. The second set consisted of Microsoft Office documents (using the Word, PowerPoint, and Excel file types) used by a product group in Microsoft for its internal product planning and development. In a similar fashion to the emails, we chose decreasing subsets of this collection as smaller file collections. To index the emails and documents, we used an indexer that employs IFilter plugins in Windows to extract unique words from each file. The indexer also extracts properties of the files from the NTFS filesystem, such as the author of a Microsoft Word document, or the artist or genre of an MP3 file.

Figure 3 shows the costs of index generation incurred by SSE, expressed as the cost per file/word pair; these are the timings for the operations that are performed after a collection of files is indexed (for the total time required to index these collections, see the results of Figure 5 in §6.3). The numbers of pairs range from about 14,000 to about 1,500,000 in number. The synthetic data is labeled with "Zipf", the Enron data is labeled with "Email", and the document data is labeled with "Docs". The annotation "(V)" labels data from executions that employed verification, and "(NV)" labels data from executions without verification. The cost per file/word pair is an amortized value: it was determined by taking the complete execution time of the each experiment and dividing by the number of file/word pairs.

The cost per file/word pair in Figure 3 is small: it decreases slowly to about 35 μs per pair without verification and about 105 μs with verification. Lower numbers of pairs lead to higher per-pair costs, since there is a constant overhead for adding new words and new files to the index, and the cost is not amortized over as many pairs in this case. For both real and synthetic data, these experiments show that adding authenticators approximately triples the cost of index generation. This is due to the extra cost of generating incremental hash values and a Merkle hash tree entry for each pair. Since CS2 uses SSE with search authenticators, we present the remainder of our micro-benchmark results only for the verifiable versions of the algorithms (however, see the full protocol results in §6.3 for a break down of the costs of SSE and search authenticators).

The email and document data validate our synthetic model and correspond closely to this model (within 10%) for data points with approximately the same number of file/word pairs. This suggests that, at least for large numbers of pairs, the Zipf model leads to the same SSE and authenticator performance as the English text as contained in the emails and documents.[6] The synthetic data tests the sensitivity of the SSE and DSA algorithms to details of the file/word distribution; experiments over the file collections are limited to always operating over the same assignment of unique words to files, but different experiments over the synthetic data contain different sets of file/word pairs, albeit drawn from the same distribution. Our results show that this sensitivity is low, as would be expected.

Execution time for some SSE and search authenticator operations do not depend on the number of file/word pairs in the index. And their cost per unique word was essentially independent (modulo a very small constant cost) of the total number of unique words (or files) in each operation. So, we present only the per-word (or per-file) time for each operation. Figure 4 shows the costs for each operation. For ease of exposition, we show numbers only for the executions of the SSE algorithm on the document data set; the numbers for the email data set and the synthetic data are similar. Search token generation takes a constant amount of time (a mean of 35 μs), irrespective of the number of files that will be returned from the search. The results show that search and file addition and deletion on the client side are efficient and practical, even for common words, or files containing many unique words.

Execution time for the remaining algorithms (authenticator generation and server addition and deletion) varies slightly in the file/word pair capacity of the index. This variability can be explained

[6] Note that the indexing code we used for the emails indexes all unique words in a file, so email headers are also indexed.

operation	time	stddev
SSE.Search/DSA.Prove	7.3	0.6
DSA.Vrfy	57	4
SSE.AddToken	37	2
SSE.DelToken	3.0	0.2

Figure 4: Execution time (in μs) per unit (word or file) for verifiable SSE operations.

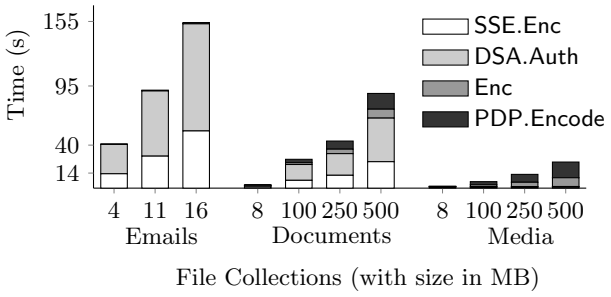

Figure 5: Execution time for STORE.

by dependence of the algorithms on the size of a proof for a Merkle hash tree, which is the logarithm of the size of the tree. So, larger trees take more computational effort on the part of the server and the client.[7]

Due to space constraints, we present only a summary of the addition and deletion graphs; see the full version of the paper for details. For our document data, SSE.Add requires approximately 12 μs per unique word to be added, and SSE.Del requires approximately 35 μs per unique word to be deleted. The difference in cost is due to the server doing most of the work for deletion, while the client performs most of the work for addition. For our document data, DSA.Add requires approximately 102 μs per unique word to be added, and DSA.Del requires approximately 109 μs per unique word to be deleted.

The execution time for file addition under verification depends on the number of words in the file that are not in the index: each new word requires the server to choose a new random location for this word entry in an authenticated data structure. Similarly, the execution time for computing the authenticator update for file deletion depends on the number of words in the file that are being removed entirely from the index. If a word will no longer exist in the index after deletion, then it is easy for the client to generate an empty replacement entry for the authenticator information; otherwise, the client must remove the word from an incremental hash function, which is computationally much more expensive. To mitigate this effect, we compared addition and deletion for the same file for all sizes of the index. Even given the variation in cost, these results demonstrate that all file addition and deletion operations are efficient and practical.

PDP. To determine the best parameters for our implementation of PDP, we ran a series of experiments over different sector sizes and sector counts (recall that the PDP tag divides a block into a fixed number of sectors). In our evaluation, we considered the costs for initial tag generation (PDP.Encode), the server side of the challenge-response protocol (PDP.Prove), and client verification of the challenge-response protocol (PDP.Vrfy). Due to space constraints, we present only the basic results; see the full version of the paper for graphs and details.

Our choice of sector size and count was based on the rate at which our implementation of the PDP algorithm can perform PDP.Encode for a given sector bit length (chosen from the values 128, 256, 384, 512, and 1024) for a randomly generated 40 MB file. All choices of the sector size (except for 1024 bits) have approximately the same performance; we chose a 256-bit sector size, since it was slightly better than the other versions, with a rate of about 35 MB/s. To minimize the size of the response in the challenge-response protocol, we chose a sector count of 100, since this is the point where the

[7] One way to mitigate this effect would be to restrict the size of trees and instead of using a single SSE index for a large data set, the client could break the data set into many small data sets and use SSE on each. Then searches, additions, and deletions could be performed linearly across the indexes. This change would make execution time for generating a search token (and the size of a search token) depend linearly on the number of SSE indexes being searched. This is a way to expand SSE indexes without using a specialized index expansion algorithm.

256-bit line begins to plateau in our experiments. We also ran other micro-benchmark experiments to confirm that this choice does not degrade the performance of PDP.Prove and PDP.Vrfy significantly compared to other choices.

6.3 CS2 Performance

To evaluate the performance of CS2 as a whole, we ran the CS2 algorithms specified in Section 4 on the email and document data sets. Additionally, we included a third data set to evaluate the performance of CS2 on media files, which have almost no indexable words but have significant file size. This collection is composed of MP3, video, WMA, and JPG files that make data sets of the same sizes as the ones in the document collection. Note that all algorithms displayed on the graphs have non-zero cost, but in some cases, the cost is so small compared to the cost of other parts of the operation that this cost cannot be seen on the graph.

Figure 5 shows the results of the STORE operation, which takes the most time of any of the algorithms.[8] Note that the entire STORE protocol is performed in addition to indexing that must be executed by the client before the data can be stored. So, all the costs of STORE are overhead for CS2 when compared to storing plain-text indexes.

Figure 5 demonstrates the difference between the email data and the document data. The Enron emails are a collection of plain text files, including email headers. This means that almost every byte of the files is part of a word that will be indexed. So, each small file contains many words, and the ratio of file/word pairs to the size of the data set is high. By contrast, Microsoft Office documents may contain significant formatting and visual components (like images) which are not indexed. So, the ratio of file/word pairs to file size is much lower. Both data sets represent a common case for office use: our results show that index generation in CS2 requires significantly more time for large text collections than for the common office document formats. Finally, the ratio of indexable words to file size is almost zero for media files. These files show the same PDP costs as the document data, but almost no SSE or authenticator costs. Note that the cost for encrypting the files depends only on the file size.

The results of Figure 5 also support the micro-benchmark results for index generation; the cost of generating a verifiable SSE index is approximately three times the cost of generating the index itself.[9] The micro-benchmark results of Figure 3 show that SSE index generation performance is linear in the number of file/word pairs for large data sets. So, for an email data set of size 16 GB (consisting entirely of text-based emails: i.e., emails containing no attachments), the initial indexing costs would be approximately 40 hours (which could be performed over the course of a few days during the idle time of the computer). After this initial indexing, adding and removing emails would be fast. And generating an index without verification for a 16 GB text-only email collection would take about 15 hours.

The CHECK operation always operates on 128 blocks chosen randomly from the collection of files, so its performance is consistent across all file collections. Figure 6 shows the costs, which are approximately 12-13 ms in each case. Most of this cost is borne by the server, which must perform $O(s\lambda)$ operations, where s is the number of sectors, and λ is the security parameter; the client only performs $O(s + \lambda)$ operations. This cost is independent of the file size, since the number of blocks checked by the algorithm is set to $\lambda = 128$ for any file collection.

[8] For clarity in this and subsequent figures, we do not present the sample standard deviations of these experiments, since they obscure the stacked bars. But the standard deviations of these results are small enough to not affect our discussion.

[9] The SSE and authenticator costs are not always trivial to separate in our implementation, since the authenticator code is tightly integrated with SSE. To compute these costs, we ran the non-verifiable SSE algorithm to get the cost of SSE alone and subtracted this SSE cost from the verifiable SSE algorithm to get the additional cost of using search authenticators.

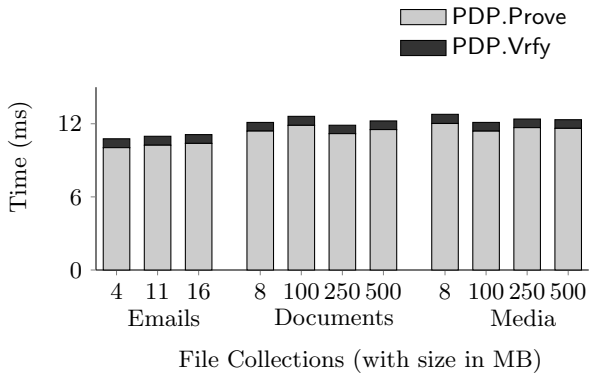

Figure 6: Execution time for CHECK.

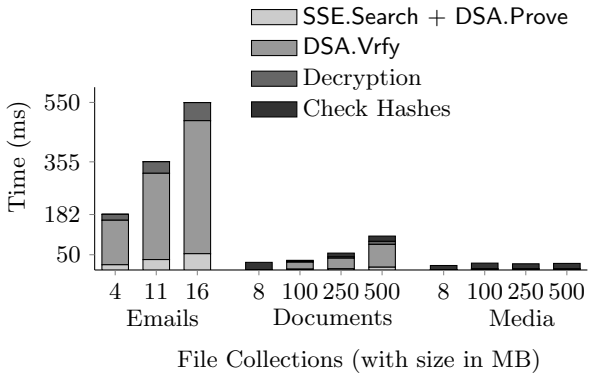

Figure 7: Execution time for SEARCH.

To evaluate the costs of SEARCH, ADD, and DELETE, we performed experiments that gave upper bounds on the cost of any operation. An upper bound for SEARCH is a search for the word contained in the most files. For ADD and DELETE, we considered operations on the file that occupied the most bytes on disk.[10]

Since SEARCH was performed on the word that was indexed for the most files, the total time needed for the search depended on the prevalence of words in files: media files had few words, even in 500 MB of content, whereas some words occur in every email. Figure 7 shows the results. The results labeled "SSE.Search + DSA.Prove" give the time needed for the server to perform a search and collect Merkle hash tree fragments for each file, given a search token (we neglect the cost of generating a search token, since it is a small constant in the microseconds). The results labeled "DSA.Vrfy" give the time needed for a client to verify the results returned by the server. The results labeled "Decryption" show the time needed to decrypt the file identifiers, and the results labeled "Check Hashes" show the time needed to check the hashes of the three files that we chose for the second round of the search. The SSE search costs were small, even for the email index; verification time dominated the total execution time of the algorithm (and all search verification is performed on the client). However, even the longest searches with verification took only about half a second to complete and verify. And for large media collections, the search and verification time was negligibly small. The Decryption and Check Hashes costs are small enough to be negligible in practice.

Figure 8 shows the execution time for the ADD protocol. The cost of the ADD operation is divided into several components: "Enc" refers to the time needed to encrypt the new file, "PDP.Encode" refers to generation of the PDP tags for the file, "SSE.AddToken" refers to client generation of the add token for the words being indexed in the file, "SSE.Add" refers to the server using the add token to update the index, "DSA.Add" refers to server collection of authenticator information for DSA update, and "DSA.Update" refers to client generation of an update to the authenticator information for the SSE index. We do not show the cost (in ADD or DELETE) for writing updates to server data structures or generating the DSA add token, since these costs are negligibly small.[11] The costs of ADD fall mostly on the client: the dominant costs are PDP tag generation, SSE add token generation, file encryption, and DSA authenticator generation, all performed on the client. In a use case where add operations dominate (such as indexing encrypted emails), this allows the server to support many clients easily, since the client that performs the add also performs most of the computations. The only server costs

[10]The other option is to perform operations on the file that contains the most words. We decided to use the number of bytes because of the relatively high costs of performing PDP tag generation. Of course, the files with the most bytes tend to have a large number of words, as well.

[11]Our current measurements overstate the server computation costs somewhat in DSA.Add by counting the cost of Merkle tree fragment verification along with the cost of tree update. This is also true of our measurements for DELETE.

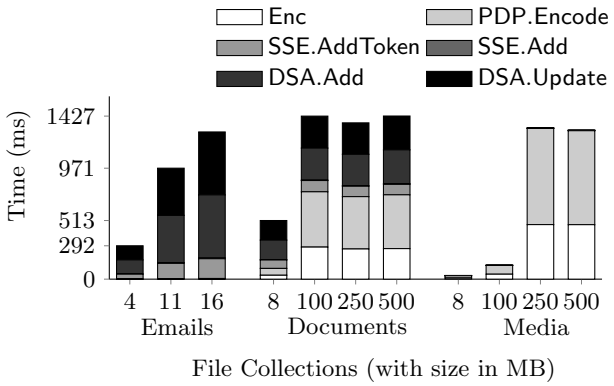

Figure 8: Execution time for ADD.

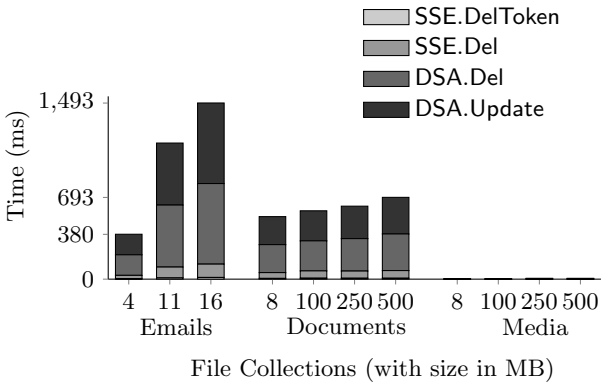

Figure 9: Execution time for DELETE.

shown on the graph is SSE.Add and DSA.Add, which are essentially negligible compared to the other costs.

A similar situation occurs in Figure 9 for the DELETE operation. The label "SSE.DelToken" refers to client generation of the delete token (this token is also used by the DSA algorithm), "SSE.Del" refers to the server using the delete token to update the index, "DSA.Del" refers to server collection of authenticator information to return to the client for DSA update, and "DSA.Update" refers to client generation of an update to the authenticator information for the SSE index. As for ADD, the DELETE operation is efficient and practical; each operation on the largest files took less than one second. No PDP costs are shown for delete, since the file is deleted without modifying the PDP tags on the server.

Acknowledgements

Thanks to Jason Mackay for writing the indexer that we used in our experiments. The first author is grateful to Kristin Lauter for her enthusiastic support of this project and numerous discussions on cryptographic cloud storage over the last three years.

References

[1] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. M. Lee, G. Neven, P. Paillier, and H. Shi. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. In V. Shoup, editor, *Advances in Cryptology – CRYPTO '05*, volume 3621 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2005.

[2] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In P. Ning, S. De Capitani di Vimercati, and P. Syverson, editors, *ACM Conference on Computer and Communication Security (CCS '07)*. ACM Press, 2007.

[3] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *Advances in Cryptology - ASIACRYPT '09*, volume 5912 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2009.

[4] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks (SecureComm '08)*, pages 1–10, New York, NY, USA, 2008. ACM.

[5] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96, Proc. 16th Annual Cryptology Conference*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15, Berlin, 1996. Springer-Verlag.

[6] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: incrementality at reduced cost. In *Advances in Cryptology - EUROCRYPT 1997*, pages 163–192. Springer-Verlag, 1997.

[7] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Advances in Cryptology – Eurocrypt'97, Proc. 16th International Conference on the Theory and Applications of Cryptographic Techniques*, volume 1233 of *Lecture Notes in Computer Science*, pages 163–192, Berlin, 1997. Springer-Verlag.

[8] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[9] D. Boneh, G. di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology – EUROCRYPT '04*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer, 2004.

[10] D. Boneh, E. Kushilevitz, R. Ostrovsky, and W. Skeith. Public-key encryption that allows PIR queries. In A. Menezes, editor, *Advances in Cryptology – CRYPTO '07*, volume 4622 of *Lecture Notes in Computer Science*, pages 50–67. Springer, 2007.

[11] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *Theory of Cryptography Conference (TCC '07)*, volume 4392 of *Lecture Notes in Computer Science*, pages 535–554. Springer, 2007.

[12] K. Bowers, A. Juels, and A. Oprea. HAIL: a high-availability and integrity layer for cloud storage. In *ACM Conference on Computer and communications security (CCS '09)*, pages 187–198. ACM, 2009.

[13] X. Boyen and B. Waters. Anonymous hierarchical identity-based encryption (without random oracles). In *Advances in Cryptology - CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 290–307. Springer, 2006.

[14] R. Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology*, 13(1), 2000.

[15] Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security (ACNS '05)*, volume 3531 of *Lecture Notes in Computer Science*, pages 442–455. Springer, 2005.

[16] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[17] Cryptography api: Next generation (windows). See `http://msdn.microsoft.com/en-us/library/aa376210(VS.85).aspx`.

[18] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.

[19] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. *Journal of Computer Security (to appear)*, 2011. See `http://eprint.iacr.org/2006/210`.

[20] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2009.

[21] Enron email dataset. See `http://www.cs.cmu.edu/~enron/`, 2009.

[22] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *ACM conference on Computer and communications security (CCS '09)*, pages 213–222, New York, NY, USA, 2009. ACM.

[23] FIPS 180-3. Secure Hash Standard (SHS). Federal Information Processing Standard (FIPS), Publication 180-3, National Institute of Standards and Technology, October 2008.

[24] FIPS 186-1. Digital Signature Standard (DSS). Federal Information Processing Standard (FIPS), Publication 186-1, National Institute of Standards and Technology, December 1998.

[25] FIPS 197. Advanced Encryption Standard (AES). Federal Information Processing Standard (FIPS), Publication 197, National Institute of Standards and Technology, November 2001.

[26] K. Fu, S. Kamara, and T. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Network and Distributed System Security Symposium (NDSS 2006)*. The Internet Society, 2006.

[27] J. Gantz and D. Reinsel. The digital universe decade - are you ready? `http://idcdocserv.com/925`, 2010.

[28] J. Gantz, D. Reinsel, C. Chite, W. Sclichting, J. McArthur, S. Minton, I. Xheneti, A. Toncheva, and A. Manfrediz. The expanding digital universe. `http://emc.com/collateral/analyst-reports/expanding-digital-idc-white-paper.pdf`, 2007.

[29] D. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole. Semantic file systems. In *ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 16–25, 1991.

[30] E-J. Goh. Secure indexes. Technical Report 2003/216, IACR ePrint Cryptography Archive, 2003. See `http://eprint.iacr.org/2003/216`.

[31] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Network and Distributed System Security Symposium (NDSS '03)*, 2003.

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

[33] M. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *Information Security Conference (ISC '08)*, volume 5222 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2008.

[34] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 265–278, 1999.

[35] W. Hörmann and G. Derflinger. Rejection-inversion to generate variates from monotone discrete distributions. *ACM T. Model. Comput. S.*, 6(3):169–184, 1996.

[36] A. Juels and B. Kaliski. PORs: Proofs of retrievability for large files. In P. Ning, S. De Capitani di Vimercati, and P. Syverson, editors, *ACM Conference on Computer and Communication Security (CCS '07)*. ACM, 2007.

[37] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *USENIX Conference on File and Storage Technologies (FAST '03)*, pages 29–42. USENIX Association, 2003.

[38] S. Kamara, C. Papamanthou, and T. Roeder. CS2: A searchable cryptographic cloud storage system. Technical Report MSR-TR-2011-58, Microsoft Research, 2011. `http://research.microsoft.com/apps/pubs/?id=148632`.

[39] V. Kher and Y. Kim. Securing distributed storage: challenges, techniques, and systems. In *ACM workshop on Storage security and survivability (StorageSS '05)*, pages 9–25. ACM, 2005.

[40] A. Leung, A. Parker-Wood, and E. L. Miller. Copernicus: A scalable, high-performance semantic file system. Technical Report CSC-SSRC-09-06, UC Santa Cruz, 2009.

[41] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[42] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Principles of distributed computing (PODC '02)*, pages 108–117, 2002.

[43] R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology – CRYPTO'87, Proc. 7th Annual Cryptology Conference*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, Berlin, 1987. Springer-Verlag.

[44] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proc. 7th USENIX Security Symposium*, pages 217–228, Berkeley, 1998.

[45] M. Naor and G. Rothblum. The complexity of online memory checking. In *IEEE Symposium on Foundations of Computer Science (FOCS '05)*, pages 573–584. IEEE Computer Society, 2005.

[46] C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *Proc. Int. Conference on Information and Communications Security (ICICS)*, volume 4861 of *LNCS*, pages 1–15. Springer, 2007.

[47] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 437–448. ACM, October 2008.

[48] R. Ada Popa, J. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with cloudproof. Technical Report MSR-TR-2010-46, Microsoft Research, 2010.

[49] M. I. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Workshop on Hot Topics in Operating Systems (HotOS '09)*. USENIX Association, 2009.

[50] H. Shacham and B. Waters. Compact proofs of retrievability. In *Advances in Cryptology - ASIACRYPT '08*. Springer, 2008.

[51] E. Shi, J. Bethencourt, T. Chan, D. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *IEEE Symposium on Security and Privacy*, pages 350–364, Washington, DC, USA, 2007. IEEE Computer Society.

[52] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: verification for untrusted cloud storage. In *ACM Workshop on Cloud Computing Security Workshop (CCSW '10)*, pages 19–30. ACM, 2010.

[53] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Research in Security and Privacy*, pages 44–55. IEEE Computer Society, 2000.

[54] B. Waters, D. Balfanz, G. Durfee, and D. Smetters. Building an encrypted and searchable audit log. In *Network and Distributed System Security Symposium (NDSS '04)*. The Internet Society, 2004.

[55] G. K. Zipf. *Psycho-Biology of Languages*. Houghton-Mifflin, Boston, 1935.

A A Dynamic Searchable Symmetric Encryption Scheme

Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme and $F : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$, $G : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^*$, $P : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$ and $Q : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^*$ be pseudo-random functions. Let $H_1 : \{0,1\}^* \to \{0,1\}^*$ and $H_2 : \{0,1\}^* \to \{0,1\}^*$ be random oracles. Let $z \in \mathbb{N}$ be the initial size of the free list. Construct a dynamic SSE scheme $\mathsf{SSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{SrchToken}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Search}, \mathsf{Add}, \mathsf{Del}, \mathsf{Dec})$ as follows:

- $\mathsf{Gen}(1^k)$: sample four k-bit strings K_1, K_2, K_3, K_4 uniformly at random and generate $K_5 \leftarrow \mathsf{SSE.Gen}(1^k)$. Output $K = (K_1, K_2, K_3, K_4, K_5)$.
- $\mathsf{Enc}(K, \mathbf{f})$:
  1. let $\mathtt{A}_s$ and $\mathtt{A}_d$ be arrays of size $|\mathbf{c}|/8 + z$ and let $\mathtt{T}_s$ and $\mathtt{T}_d$ be lookup tables of size $\#W$ and $\#\mathbf{f}$, respectively. We assume $\mathbf{0}$ is a $(\log \#\mathtt{A})$-length string of 0's and free is a string not in W.
  2. for each word $w \in W$,
     (a) create a list $\mathtt{L}_w$ of $\#\mathbf{f}_w$ nodes $(\mathtt{N}_1, \ldots, \mathtt{N}_{\#\mathbf{f}_w})$ stored at random locations in the search array $\mathtt{A}_s$ and defined as:

$$\mathtt{N}_i := \left(\langle id_i, \mathsf{addr}_s(\mathtt{N}_{i-1}), \mathsf{addr}_s(\mathtt{N}_{i+1}) \rangle \oplus H_1(K_w, r_i), r_i \right)$$

     where id_i is the ID of the ith file in $\mathbf{f}_w$, r_i is a k-bit string generated uniformly at random, $K_w := P_{K_3}(w)$ and $\mathsf{addr}_s(\mathtt{N}_{\#\mathbf{f}_w+1}) = \mathsf{addr}_s(\mathtt{N}_0) = \mathbf{0}$

     (b) store a pointer to the first node of $\mathtt{L}_w$ in the search table by setting

$$\mathtt{T}_s[F_{K_1}(w)] := \langle \mathsf{addr}_s(\mathtt{N}_1), \mathsf{addr}_d(\mathtt{N}_1^\star) \rangle \oplus G_{K_2}(w),$$

     where $\mathtt{N}^\star$ is the dual of $\mathtt{N}$, i.e., the node in $\mathtt{A}_d$ whose fourth entry points to $\mathtt{N}_1$ in $\mathtt{A}_s$.

3. for each file f in $\mathbf{f}$,
   (a) create a list $\mathtt{L}_f$ of $\#f$ dual nodes $(\mathtt{D}_1, \ldots, \mathtt{D}_{\#f})$ stored at random locations in the deletion array $\mathtt{A}_d$ and defined as:

$$\mathtt{D}_i := \Big(\big\langle \mathsf{addr}_d(\mathtt{D}_{i+1}), \mathsf{addr}_d(\mathtt{N}^{\star}_{j-1}), \mathsf{addr}_d(\mathtt{N}^{\star}_{j+1}), \mathsf{addr}_s(\mathtt{N}_j), \mathsf{addr}_s(\mathtt{N}_{j-1}), \mathsf{addr}_s(\mathtt{N}_{j+1}), F_{K_1}(w)\big\rangle \oplus H_1(K_f, r'_i), r'_i\Big)$$

   where j is the index of the node in list $\mathtt{L}_{w_i} = (\mathtt{N}_1, \ldots, \mathtt{N}_{\#\mathbf{f}_{w_i}})$ for the file f, r'_i is a k-bit string generated uniformly at random, $K_f := P_{K_3}(f)$, and $\mathsf{addr}_d(\mathtt{D}_{\#f+1}) = \mathsf{addr}_d(\mathtt{D}_0) = \mathbf{0}$.
   (b) store a pointer to the first node of $\mathtt{L}_f$ in the deletion table by setting:

$$\mathtt{T}_d[F_{K_1}(f)] := \mathsf{addr}_d(\mathtt{D}_1) \oplus G_{K_2}(f)$$

4. create the free list $\mathtt{L}_{\mathsf{free}}$ by choosing z unused cells at random in $\mathtt{A}_s$ and in $\mathtt{A}_d$. Let $(\mathcal{F}_1, \ldots, \mathcal{F}_z)$ and $(\mathcal{F}'_1, \ldots, \mathcal{F}'_z)$ be the free nodes in $\mathtt{A}_s$ and $\mathtt{A}_d$, respectively. Set

$$\mathtt{T}_s[F_{K_1}(\mathsf{free})] := \langle \mathsf{addr}_s(\mathcal{F}_z), \mathbf{0}^{\log \#\mathtt{A}} \rangle \oplus G_{K_2}(\mathsf{free})$$

   and for $z \leq i \leq 1$, set

$$\mathtt{A}_s[\mathsf{addr}_s(\mathcal{F}_i)] := \langle \mathbf{0}^{\log \#\mathbf{f}}, \mathsf{addr}_s(\mathcal{F}_{i-1}), \mathsf{addr}_d(\mathcal{F}^{\star}_i) \rangle \oplus Q_{K_4}(i)$$

   where $\mathsf{addr}_s(\mathcal{F}_0) = \mathbf{0}^{\log \#\mathtt{A}}$.
5. fill the remaining entries of $\mathtt{A}_s$ and $\mathtt{A}_d$ with random strings
6. for $1 \leq i \leq \#\mathbf{f}$, let $c_i \leftarrow \mathsf{SKE.Enc}_{K_5}(f_i)$
7. output $(\gamma, \mathbf{c})$, where $\gamma := (\mathtt{A}_s, \mathtt{T}_s, \mathtt{A}_d, \mathtt{T}_d)$ and $\mathbf{c} = (c_1, \ldots, c_{\#\mathbf{f}})$.

- $\mathsf{SrchToken}(K, w)$: compute and output $\tau_s := \big(F_{K_1}(w), G_{K_2}(w), P_{K_3}(w)\big)$
- $\mathsf{Search}(\gamma, \mathbf{c}, \tau_s)$:
  1. parse τ_s as (τ_1, τ_2, τ_3) and recover a pointer to the first node of the list by computing $(\alpha_1, \alpha'_1) := \mathtt{T}_s[\tau_1] \oplus \tau_2$
  2. lookup node $\mathtt{N}_1 := \mathtt{A}[\alpha_1]$ and decrypt it using τ_3, i.e., parse $\mathtt{N}_1$ as (ν_1, r_1) and compute $(id_1, \mathbf{0}, \mathsf{addr}_s(\mathtt{N}_2)) := \nu_1 \oplus H_1(\tau_3, r_1)$
  3. for $i \geq 2$, decrypt node $\mathtt{N}_i$ as above until $\alpha_{i+1} = \mathbf{0}$
  4. let $I = \{id_1, \ldots, id_m\}$ be the file identifiers revealed in the previous steps and output $\{c_i\}_{i \in I}$, i.e., the encryptions of the files whose identifiers was revealed.
- $\mathsf{AddToken}(K, f)$: parse f as $(w_1, \ldots, w_{\#f})$ and output

$$\tau_a := \Big(F_{K_1}(\mathsf{free}), G_{K_2}(\mathsf{free}), P_{K_3}(f), \#f, \lambda_1, \ldots, \lambda_{\#f}, c\Big),$$

  where $c \leftarrow \mathsf{SKE.Enc}_{K_5}(f)$ and for all $1 \leq i \leq \#f$:

$$\lambda_i := \Big(F_{K_1}(w_i), G_{K_2}(w_i), Q_{K_4}(\#\mathtt{L}_{\mathsf{free}} + 1 - i), \langle id(f), \mathbf{0}, \mathbf{0} \rangle \oplus H_1\big(P_{K_3}(w_i), r_i\big), r_i\Big),$$

  and r_i is a random k-bit string. Update $\#\mathtt{L}_{\mathsf{free}}$ to $\#\mathtt{L}_{\mathsf{free}} - \#f$.

- $\mathsf{Add}(\gamma, \mathbf{c}, \tau_a)$:

  1. parse τ_a as $(\tau_1, \ldots, \tau_4, \lambda_1, \ldots, \lambda_{\#f}, c)$ and for $1 \leq i \leq \#f$,
     (a) find the last free location in the search array by computing $\langle \varphi, \mathbf{0}^{\log \#\mathtt{A}} \rangle := \mathtt{T}_s[\tau_1] \oplus \tau_2$
     (b) recover the location of the second to last free entry by computing $(\varphi_{-1}, \varphi^\star) := \mathtt{A}_s[\varphi] \oplus \lambda_1[3]$
     (c) update the search table to point to the second to last free entry by setting $\mathtt{T}_s[\tau_1] := \varphi_{-1} \oplus \tau_2$
     (d) recover a pointer to the first node $\mathtt{N}_1$ of the list by computing $(\alpha_1, \alpha_1^\star) := \mathtt{T}_s[\lambda_i[1]] \oplus \lambda_i[2]$
     (e) make $\mathtt{N}_1$'s back pointer point to the new node by setting: $\mathtt{A}_s[\alpha_1] := \big(\mathtt{N}_1 \oplus \langle \mathbf{0}, \varphi, \mathbf{0} \rangle, r\big)$, where $(\mathtt{N}_1, r) := \mathtt{A}[\alpha_1]$
     (f) store the new node at location φ and modify its forward pointer to $\mathtt{N}_1$ by setting $\mathtt{A}_s[\varphi] := \big(\lambda_i[4] \oplus \langle \mathbf{0}, \mathbf{0}, \alpha_1 \rangle, \lambda_i[5]\big)$
     (g) update the search table by setting $\mathtt{T}_s[\lambda_i[1]] := (\varphi, \varphi^\star) \oplus \lambda_i[2]$
     (h) update the dual of $\mathtt{N}_1$ by setting:

$$\mathtt{A}_d[\alpha_1^\star] := \big(\mathtt{D}_1 \oplus \langle \mathbf{0}, \varphi^\star, \mathbf{0}, \mathbf{0}, \varphi, \mathbf{0}, \mathbf{0} \rangle, r\big),$$

     where $(\mathtt{D}_1, r) := \mathtt{A}_d[\alpha_1^\star]$
     (i) update the dual of $\mathtt{A}_s[\varphi]$ by setting:

$$\mathtt{A}_d[\varphi^\star] := \big(\langle \varphi_{-1}^\star, \mathbf{0}, \alpha^\star, \alpha, \mathbf{0}, \varphi, \lambda_i[1] \rangle \oplus H_2(\tau_3, r), r\big),$$

     where r is a random k-bit string.
  2. update the ciphertexts by adding c to $\mathbf{c}$

- $\mathsf{DelToken}(K, f)$: output:

$$\tau_d := \Big(F_{K_1}(f), G_{K_2}(f), P_{K_3}(f), F_{K_1}(\mathsf{free}), G_{K_2}(\mathsf{free}), Q_{K_4}(\#\mathtt{L}_{\mathsf{free}}{+}1), \ldots, Q_{K_4}(\#\mathtt{L}_{\mathsf{free}}{+}\#f), \#f, id(f)\Big).$$

  Update $\#\mathtt{L}_{\mathsf{free}}$ to $\#\mathtt{L}_{\mathsf{free}} + \#f$.

- $\mathsf{Del}(\gamma, \mathbf{c}, \tau_d)$:

  1. parse τ_d as $(\tau_1, \ldots, \tau_{5+\#f}, \#f, id)$
  2. find the first node of $\mathtt{L}_f$ by computing $\alpha_1' := \mathtt{T}_d[\tau_1] \oplus \tau_2$
  3. for $1 \leq i \leq \#\mathbf{f}$,
     (a) decrypt $\mathtt{D}_i$ by computing $(\alpha_1, \ldots, \alpha_6, \mu) := \mathtt{D}_i \oplus H_2(\tau_3, r)$, where $(\mathtt{D}_i, r) := \mathtt{A}_d[\alpha_i']$
     (b) delete $\mathtt{D}_i$ by setting $\mathtt{A}_d[\alpha']$ to a random $(6 \log \#\mathtt{A} + k)$-bit string
     (c) find address of last free node by computing $(\varphi, \mathbf{0}^{\log \#\mathtt{A}}) := \mathtt{T}_s[\tau_4] \oplus \tau_5$
     (d) make the free entry in the search table point to $\mathtt{D}_i$'s dual by setting

$$\mathtt{T}_s[\tau_4] := \langle \alpha_4, \mathbf{0}^{\log \#\mathtt{A}} \rangle \oplus \tau_5$$

     (e) "free" location of $\mathtt{D}_i$'s dual by setting $\mathtt{A}_s[\alpha_4] := (\varphi, \alpha') \oplus \tau_{5+i}$

(f) let $\mathtt{N}_{-1}$ be the node that precedes $\mathtt{D}_i$'s dual. Update $\mathtt{N}_{-1}$'s "next pointer" by setting:

$$\mathtt{A}_s[\alpha_5] := (\beta_1, \beta_2, \beta_3 \oplus \alpha_4 \oplus \alpha_6),$$

where $(\beta_1, \beta_2, \beta_3) := \mathtt{A}_s[\alpha_5]$. Also, update the pointers of $\mathtt{N}_{-1}$'s dual by setting

$$\mathtt{A}_d[\alpha_2] := (\beta_1, \ldots, \beta_5, \beta_6 \oplus \alpha_4 \oplus \alpha_6, \mu*),$$

where $(\beta_1, \ldots, \beta_6, \mu^*) := \mathtt{A}_d[\alpha_2]$

(g) let $\mathtt{N}_{+}$ be the node that follows $\mathtt{D}_i$'s dual. Update $\mathtt{N}_{+1}$'s "previous pointer" by setting:

$$\mathtt{A}_s[\alpha_6] := (\beta_1, \beta_2 \oplus \alpha_4 \oplus \alpha_5, \beta_3),$$

where $(\beta_1, \beta_2, \beta_3) := \mathtt{A}_s[\alpha_6]$. Also, update $\mathtt{N}_{+1}$'s pointers by setting:

$$\mathtt{A}_d[\alpha_3] := (\beta_1, \ldots, \beta_4, \beta_5 \oplus \alpha_4 \oplus \alpha_5, \beta_6, \mu^*),$$

where $(\beta_1, \ldots, \beta_6, \mu^*) := \mathtt{A}_d[\alpha_3]$

(h) set $\alpha'_i := \alpha_1$

4. remove the ciphertext that corresponds to id from $\mathbf{c}$

- $\mathsf{Dec}(K, c)$: output $m := \mathsf{SKE.Dec}_{K_5}(c)$.