CS2: A Semantic Cryptographic Cloud Storage System[*]

Seny Kamara [†] Charalampos Papamanthou [‡] Tom Roeder [§]

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, government 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 to providing security against providers (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 provides provable guarantees of confidentiality, integrity, and verifiability without sacrificing utility. In particular, while CS2 provides security against the cloud provider, clients are still able to efficiently access their data through a *search* interface and to add and delete files.

This is achieved using novel cryptographic primitives and protocols for dynamic searchable symmetric encryption, authenticated data structures, and dynamic proofs of storage. Experimental results from an implementation of CS2 demonstrate its efficiency in practice.

[*]**DRAFT: 05/17/11**

[†]Microsoft Research. `senyk@microsoft.com`.

[‡]Brown University. `cpap@cs.brown.edu`. Work done as an intern at Microsoft Research.

[§]Microsoft Research. `throeder@microsoft.com`.

Contents

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 records kept by organizations that are subject to regulation such as Sarbanes-Oxley and HIPAA. 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 [GRC$^+$07, GR10], the total amount of digital data generated by consumers and enterprises will grow next year to 1.2 zettabytes, i.e., 1.2 million petabytes. The increasing scale of stored data makes it harder to store, manage, and analyze. Cloud storage services, such as Amazon S3 or Microsoft Azure Storage, offer several advantages to clients including elasticity (the ability to scale storage rapidly), universal access (the ability to access data from anywhere) and reliability. However, all of these services suffer from security and privacy concerns which have motivated research on secure cloud storage systems [GSMB03, PLM$^+$10, SCC$^+$10, BJO09, GPTT08], 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.

Semantic access. The first limitation of cryptographic storage systems is that they only allow clients to access data through file system navigation. Today, however, data is mostly accessed through search. The need for search in storage systems has long been recognized and has motivated a great deal of work on *semantic* file systems [GJSO91, GM99, LPWM09, SM09] 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.

Global integrity. 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 (1) with *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 our data, namely no recomputation from scratch on all the data is required whenever an update (e.g., deletion of a file) is issued.

Our approach. At a high level, we achieve these properties by replacing traditional cryptographic primitives like symmetric encryption and MACs or digital signatures with new cryptosystems that are better suited to the cloud storage setting. These include symmetric searchable encryption (SSE), search authenticators (a kind of authenticated data structure) 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 through the use of 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 at the time of this writing CS2 is a personal storage system, since it does not allow clients to share files. We stress, however, that the design of CS2 is flexible enough that several sharing mechanisms could be layered on top of it. We discuss this further in §8.1 and describe several possible approaches to integrate sharing.

1.1 Our contributions.

We make several contributions in this work. The first is the design of CS2, i.e., the underlying cryptographic protocol on which the system is based. To evaluate the security of our design, we prove that it emulates a trusted cloud storage system. Our approach here follows the ideal/real-world paradigm typically used to prove the security of cryptographic protocols. This leads us to our second contribution, which is our formal security definition for cloud storage systems. Our third contribution is our implementation of CS2, which we evaluate by conducting experiments on real data. The performance experiments in §7.2 show that CS2 is efficient enough to be used in practice

As mentioned above, the CS2 protocol relies on several cryptographic primitives and protocols. However, all the currently known instantiations of these primitives have limitations that make them ill-suited for our purposes. So, we designed new cryptographic schemes and considerably extended previous work. This results in (1) the first *dynamic* SSE scheme that achieves sub-linear search time (in fact, our scheme is asymptotically optimal); (2) a new kind of search authenticator that achieves stronger security properties than considered in previous work; and (3) an efficient unbounded dynamic PoS scheme.

Finally, while CS2 was designed primarily as a cloud storage system, its underlying libraries can be used to build or enhance other kinds of storage systems such as search applications and semantic file systems. We discuss some of these extensions in §8.1.

2 Related Work

Semantic file systems. Semantic file systems were first proposed and motivated by Gifford, Jouvelot, Sheldon and O'Toole [GJSO91]. In this work they describe their design and implementation of the SFS system and introduce the notion of a virtual directory: directories whose content is generated dynamically as the result of a search query. More recent work on semantic file systems includes [Gia98, GM99, XKTK03, PR03, Cor, LPWM09, SM09].

Cryptographic storage systems. 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 [Bla93, Bla94, ZBS98, MKKW99, BCW99, Fu99, CCSP01, KRS$^+$03, KRS$^+$03, GSMB03, Cor03, WMZ03, LKMS04] and we refer the reader to the survey by Kher and Kim [KK05] for further details. 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, none of the previously proposed storage systems can achieve scalable search or global integrity.

Plutus is a cryptographic file system that provides sharing on a per-file basis. It uses the concept of lazy revocation from Cepheus [Fu99] combined with a novel key distribution mechanism called key rotation to handle revocations efficiently. In [FKK06], Fu, Kamara and Kohno point out that key rotation, as described and instantiated in [KRS$^+$03], is not provably secure and propose the stronger notion of key regression. SiRiUS [GSMB03] 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).

Consistent storage systems. A consistent file system is a shared system that guarantees its users a consistent view of their data, i.e., an untrusted storage service cannot provide different versions of the data to different clients without being detected. This guarantee was introduced and formalized as fork-consistency by Mazières and Shasha in [MS02] and first implemented in SUNDR [LKMS04].

CloudProof [PLM$^+$10] is a cryptographic cloud storage system that provides read/write sharing. It achieves confidentiality, integrity, fork-consistency and freshness (for both data and meta-data). 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. Sharing and efficient revocation is achieved through a combination of broadcast encryption with lazy revocation and key rotation (see our comment above). CloudProof achieves fork-consistency by periodically executing an auditing protocol between the data owner and the authorized clients.

Venus [SCC$^+$10] is a cloud storage system that provides integrity and "eventual consistency". The latter is weaker than fork-consistency but avoids some of its limitations [CKS09b, CKS09a]. More precisely, eventual consistency guarantees that the clients' views of a shared file will eventually either

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

be consistent or will be detected as inconsistent. And unlike fork-consistency, it does not require any communication between the clients, it is "wait-free" (i.e., clients will never block on writes) and it guarantees that client operations will complete.

Distributed & verifiable storage systems. HAIL [BJO09] is a distributed cloud storage system that provides integrity and availability for static data, i.e., the client can verify at any time that its data can be retrieved. This is achieved by carefully encoding the data, distributing it across several service providers and using proofs of storage [JK07]. Athos [GPTT08] is a verifiable hierarchichal file system that can be layered on top of any cloud storage service. Using Athos, a client can store its data in the cloud and verify the correctness of any file system operation it makes (e.g., `ls`, `cd`, `mkdir`, etc...). Verifiability is achieved by requiring the cloud provider to return a short proof that its answers to the client's operations are correct.

Secure file systems. Canetti et al. describe simpfs [CCH+10], a stackable file system that can be implemented on top of any POSIX file system to provide provable security against file manipulation attacks. In such an attack, an adversary creates filesystem links to fool an application with higher privileges into opening particular files. While these concerns are orthogonal to our goals, the approach taken in [CCH+10] to analyze simpfs is similar to our own. In particular, both simpfs and CS2 are provably secure in the ideal/real-world model traditionally used to analyze the security of cryptographic protocols. Unlike CS2, however, simpfs is shown in [CCH+10] to be *universally composable*, which means that it can be used safely in any environment. In this work, on the other hand, we only show that CS2 is secure in a standalone setting (i.e., where the parties are running only a single instance of CS2).

Searchable encryption. The problem of searching on symmetrically encrypted data can be solved in its full generality using the work of Goldreich and Ostrovsky [GO96] on oblivious RAMs. In addition to handling any type of search query, this approach also provides the strongest levels of security, namely the server does not learn any information about the data or the queries—not even information inferred by the client's access pattern. Unfortunately, this approach requires interaction and has a high overhead for the server.

Searchable encryption was considered explicitly by Song, Wagner and Perrig in [SWP00], where they give a non-interactive solution that achieves search time that is linear in the length of the file collection. In [Goh03], Goh introduced formal security definitions for symmetric searchable encryption and proposed a construction based on Bloom filters [Blo70] that requires $O(n)$ search time on the server (where n is the number of files) and results in false positives. This index-based approach to constructing SSE schemes was adopted by Chang and Mitzenmacher [CM05] who proposed alternative security definitions and a new construction also with $O(n)$ search time but without false positives.

In [CGKO06], Curtmola, Garay, Kamara and Ostrovsky proposed even stronger security definitions and gave the first construction to achieve sub-linear (and in fact optimal) search time. While the approach of [CGKO06] is efficient and does not induce false positives, it does not explicitly handle dynamic data, i.e., it can be made dynamic only by using general techniques from data structures that are relatively inefficient. Starting with the work of Boneh, Di Crescenzo, Ostrovsky and Persiano [BdCOP04], searchable encryption has also been considered in the public-key setting [BdCOP04, WBDS04, ABC+05, BW06, BKOS07, BW07].

In [CK10], Chase and Kamara introduced the notion of structured encryption which generalizes index-based SSE to arbitrarily structured data. In that work, the authors give constructions for several data types and, in particular, show how to construct a graph encryption scheme that handles neighbor queries (i.e., given a node, return its neighbors) from any SSE scheme.

Proofs of storage. 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. [ABC+07]; and (2) proofs of retrievability (PoR), introduced by Juels and Kaliski [JK07]. Earlier work by Naor and Rothblum [NR05] 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 [SW08, APMT08, DVW09, BJO09, AKK09, EKPT09]. 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) [ABC+07]. In this work, we construct a dynamic PDP scheme from the efficient HLA used in Shacham and Waters' privately-verifiable PoR [SW08].

Authenticated data structures. Relevant to this work are also authenticated data structures [MND+04, Tam03] (e.g., the search authenticator used is an authenticated data structure). An authenticated data structure provides the machinery and the algorithms for efficiently verifying data and computations stored at and performed by untrusted entities. Conceptually, an authenticated data structure provides the same security properties as a memory checking protocol [NR05] but in a more efficient way. Several authenticated data structures have appeared in the literature since Naor's seminal paper [NN98] that proposed the first efficient dynamization of Merkle trees [Mer87]. Such solutions include efficient verification of queries on databases [MND+04], graph and geometric queries [GTT11], hash table operations [PTT08] and dynamic dictionary constructions [PT07].

3 Preliminaries and Notation

Notation. The set of all binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. $[n]$ is the set of integers $\{1,\ldots,n\}$. We write $x \leftarrow \chi$ to represent an element x being sampled from a distribution χ, and $x \overset{\$}{\leftarrow} X$ to represent an element x being sampled uniformly at random from a set X. The output x of a probabilistic algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$ and that of a deterministic algorithm $\mathcal{B}$ by $x := \mathcal{B}$. Given a sequence of elements $\mathbf{v}$ we refer to its i^{th} element either as v_i or $\mathbf{v}[i]$ and to its total number of elements as $\#\mathbf{v}$. If S is a set then $\#S$ refers to its cardinality. W denotes the universe of words. If $f = (w_1, \ldots, w_m) \in W^m$ is a file, then $\#f$ denotes its total number of words and $|f|$ is its bit length. If s is a string then $|s|$ refers to its bit length. We denote the concatenation of two strings s and s' by $\langle s, s' \rangle$.

We use various data structures including linked lists, arrays and lookup tables. If $\mathtt{L}$ is a list then $\#\mathtt{L}$ denotes its total number of nodes. If $\mathtt{A}$ is an array then $\#\mathtt{A}$ is its total number of cells, and if $\mathtt{T}$ is a lookup table then $\#\mathtt{T}$ is its number of non-empty entries.

Throughout, $k \in \mathbb{N}$ will denote the security parameter and we will assume all algorithms take k implicitly as input. A function $\nu : \mathbb{N} \to \mathbb{N}$ is negligible in k if for every positive polynomial $p(\cdot)$ and all sufficiently large k, $\nu(k) < 1/p(k)$. We write $f(k) = \mathsf{poly}(k)$ to mean that there exists a polynomial $p(\cdot)$ such that $f(k) \leq p(k)$ for all sufficiently large $k \in \mathbb{N}$; and similarly write $f(k) = \mathsf{negl}(k)$ to mean that there exists a negligible function $\nu(\cdot)$ such that $f(k) \leq \nu(k)$ for all sufficiently large k. Two distributions ensembles χ and χ' are computationally indistinguishable if for all probabilistic polynomial-time distinguishers $\mathcal{D}$,

$$\left|\Pr\left[\,\mathcal{D}(\chi) = 1\,\right] - \Pr\left[\,\mathcal{D}(\chi') = 1\,\right]\right| \leq \mathsf{negl}(k).$$

We sometimes write this as $\chi \overset{c}{\approx} \chi'$.

Basic cryptographic primitives. A private-key encryption scheme is a set of three polynomial-time algorithms $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ such that Gen is a probabilistic algorithm that takes a security parameter k and returns a secret key K; Enc is a probabilistic algorithm takes a key K and a message m and returns a ciphertext c; Dec is a deterministic algorithm that takes a key K and a ciphertext c and returns m if K was the key under which c was produced. Informally, a private-key encryption scheme is CPA-secure if the ciphertexts it outputs do not reveal any partial information about the plaintext even to an adversary that can adaptively query an encryption oracle.

In addition to encryption schemes, we also make use of pseudo-random functions (PRF) and permutations (PRP), which are polynomial-time computable functions that cannot be distinguished from random functions by any probabilistic polynomial-time adversary. We refer the reader to [KL08] for formal definitions of CPA-security, PRFs and PRPs.

Incremental hash functions. An incremental hash function [BGG94] is a collision-resistant function $\mathrm{IH} : \{0,1\}^* \rightarrow \{0,1\}^\ell$ for which there exists two efficiently computable operations (which we denote by addition and subtraction) such that: (1) given two strings $\mathrm{IH}(x)$ and y, where $x, y \in \{0,1\}^*$, one can compute $\mathrm{IH}(\langle x, y\rangle) := \mathrm{IH}(x) + y$; and (2) given strings $\mathrm{IH}(\langle x, y\rangle)$ and y one can compute $\mathrm{IH}(x) := \mathrm{IH}(\langle x, y\rangle) - y$. Several efficient incremental hash functions were proposed by Bellare and Micciancio [BM97a].

Merkle trees. A Merkle tree [Mer89] allows one to hash a set of n elements using a a *single* hash value in such a way that the integrity of any individual element can be verified using only a logarithmic number of hashes. Given a set of n elements $X = \{x_1, \ldots, x_n\}$ one places the elements at the leaves of a (complete) binary tree (here we assume n is a power of 2) and sets each internal node of the tree to be the hash of its children. Given the hash at the root of the tree, one can verify the integrity of any element in X given an *authentication path* which consists of the path from the element to the root together with the siblings of any node in path.

4 Cryptographic Building Blocks

We describe the cryptographic building blocks used in CS2, including symmetric searchable encryption, search authenticators and proofs of storage.

4.1 Searchable Symmetric 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 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 constructions [Goh03, CM05, CGKO06] 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 and given τ_w, γ and $\mathbf{c}$, the provider can find the subset of ciphertexts $\mathbf{c}_w \subseteq \mathbf{c}$ that contain 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 [GO96], 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 and given τ and γ, the provider can update the encrypted index γ. Similarly, to delete a file f, the client generates a delete token τ', which the provider uses to update γ.

Definition 4.1 (Dynamic SSE)**.** *A dynamic index-based searchable symmetric encryption scheme is a tuple of nine polynomial-time algorithms* $\mathsf{SSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{SrchToken}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Search}, \mathsf{Add}, \mathsf{Del}, \mathsf{Dec})$ *such that:*

> $K \leftarrow \mathsf{Gen}(1^k)$*: is a probabilistic algorithm that takes as input a security parameter k and outputs a secret key K.*
>
> $(\gamma, \mathbf{c}) \leftarrow \mathsf{Enc}(K, \delta, \mathbf{f})$*: is a probabilistic algorithm that takes as input a secret key K, an index δ, and a sequence of files $\mathbf{f}$. It outputs an encrypted index γ and a sequence of ciphertexts $\mathbf{c}$.*
>
> $\tau_s \leftarrow \mathsf{SrchToken}(K, w)$*: is a (possibly probabilistic) algorithm that takes as input a secret key K and a keyword w and outputs a search token τ_s.*
>
> $\tau_a \leftarrow \mathsf{AddToken}(K, f)$*: is a (possibly probabilistic) algorithm that takes as input a secret key K and a file f, and outputs an add token τ_a.*
>
> $\tau_d \leftarrow \mathsf{DelToken}(K, id)$*: is a (possibly probabilistic) algorithm that takes as input a secret key K and a file id, and outputs a delete token τ_d.*
>
> $\mathbf{c}_w := \mathsf{Search}(\gamma, \mathbf{c}, \tau_s)$*: is a deterministic algorithm that takes as input an encrypted index γ, a sequence of ciphertexts $\mathbf{c}$ and a search token τ_s. It outputs a sequence of ciphertexts $\mathbf{c}_w \subseteq \mathbf{c}$.*
>
> $(\gamma', \mathbf{c}') := \mathsf{Add}(\gamma, \mathbf{c}, \tau_a)$*: is a deterministic algorithm that takes as input an encrypted index γ, a sequence of ciphertexts $\mathbf{c}$, and an add token τ_a. It outputs a new encrypted index γ' and new sequence of ciphertexts $\mathbf{c}'$.*
>
> $(\gamma', \mathbf{c}') := \mathsf{Del}(\gamma, \mathbf{c}, \tau_d)$*: is a deterministic algorithm that takes as input an encrypted index γ, a sequence of ciphertexts $\mathbf{c}$, and a delete token τ_d. It outputs a new encrypted index γ' and new sequence of ciphertexts $\mathbf{c}'$.*

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

$f := \mathsf{Dec}(K, c)$: *is a deterministic algorithm that takes as input a secret key K and a ciphertext c and outputs a file f.*

Intuitively, the security guarantee we require from a dynamic SSE scheme is that that (1) given an encrypted index γ and a sequence of ciphertexts $\mathbf{c}$, no adversary can learn any partial information about the files $\mathbf{f}$; and that (2) given, in addition, a sequence of tokens $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_n)$ for an adaptively generated sequence of queries $\mathbf{q} = (q_1, \ldots, q_n)$ (which can be for the search, add or delete operations), no adversary can learn any partial information about either $\mathbf{f}$ or $\mathbf{q}$.

This exact intuition can be difficult to achieve and most known efficient and non-interactive SSE schemes [Goh03, CM05, CGKO06] reveal the access and search patterns. [3] We therefore need to weaken the definition appropriately by allowing some limited information about the messages and the queries to be revealed to the adversary. To capture this, we follow the approach of [CGKO06] and [CK10] and parameterize our definition with a set of leakage functions that capture precisely what is being leaked by the ciphertext and the tokens.

As observed in [CGKO06], another issue with respect to SSE security is whether the scheme is secure against *adaptive* chosen-keyword attacks (CKA2) or only against *non-adaptive* chosen keyword attacks (CKA1). The former guarantees security even when the client's queries are based on the encrypted index and the results of previous queries. The latter only only guarantees security if the client's queries are independent of the index and of previous results.

In the following definition, we extend the notion of CKA2-security from [CGKO06] to the setting of dynamic SSE.

Definition 4.2 (CKA2-security)**.** *Let* $\mathsf{SSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{SrchToken}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Search}, \mathsf{Add}, \mathsf{Del}, \mathsf{Dec})$ *be a dynamic private-key index-based SSE scheme and consider the following probabilistic experiments, where $\mathcal{A}$ is a stateful adversary, $\mathcal{S}$ is a stateful simulator and $\mathcal{L}_1^{\mathsf{sse}}$, $\mathcal{L}_2^{\mathsf{sse}}$, $\mathcal{L}_3^{\mathsf{sse}}$ and $\mathcal{L}_4^{\mathsf{sse}}$ are stateful leakage algorithms:*

$\mathbf{Real}_{\mathcal{A}}^{\mathsf{sse}}(k)$: *the challenger runs $\mathsf{Gen}(1^k)$ to generate a key K. $\mathcal{A}$ outputs a tuple $(\delta, \mathbf{f})$ and receives $(\gamma, \mathbf{c}) \leftarrow \mathsf{Enc}_K(\delta, \mathbf{f})$ from the challenger. The adversary makes a polynomial number of adaptive queries $q \in \{w, f, id\}$ and, for each query q, receives from the challenger either a search token $\tau_s \leftarrow \mathsf{SrchToken}_K(w)$, an add token $\tau_a \leftarrow \mathsf{AddToken}_K(f)$ or a delete token $\tau_d \leftarrow \mathsf{DelToken}_K(id)$. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\mathsf{sse}}(k)$: *$\mathcal{A}$ outputs a tuple $(\delta, \mathbf{f})$. Given $\mathcal{L}_1^{\mathsf{sse}}(\delta, \mathbf{f})$, $\mathcal{S}$ generates and sends a pair $(\gamma, \mathbf{c})$ to $\mathcal{A}$. The adversary makes a polynomial number of adaptive queries $q \in \{w, f, id\}$ and, for each query q, the simulator is given either $\mathcal{L}_2^{\mathsf{sse}}(\delta, \mathbf{f}, w)$, $\mathcal{L}_3^{\mathsf{sse}}(\delta, \mathbf{f}, f)$ or $\mathcal{L}_4^{\mathsf{sse}}(\delta, \mathbf{f}, id)$. The simulator returns an appropriate token τ. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.*

We say that sse *is $(\mathcal{L}_1^{\mathsf{sse}}, \mathcal{L}_2^{\mathsf{sse}}, \mathcal{L}_3^{\mathsf{sse}}, \mathcal{L}_4^{\mathsf{sse}})$-secure against adaptive dynamic chosen-keyword attacks if for all* PPT *adversaries $\mathcal{A}$, there exists a* PPT *simulator $\mathcal{S}$ such that*

$$|\Pr\left[\mathbf{Real}_{\mathcal{A}}^{\mathsf{sse}}(k) = 1\right] - \Pr\left[\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\mathsf{sse}}(k) = 1\right]| \leq \mathsf{negl}(k).$$

Leakage. As in previous work, the $\mathcal{L}_2^{\mathsf{sse}}$ leakage of our construction consists mainly of the access and search patterns. However, because the scheme is dynamic there is additional leakage that occurs on updates. We refer to this leakage as the word pattern. Intuitively, it reveals for a (unknown) search term w if and when a file containing w was either added or deleted in the past.

[3] Two exceptions are the work of Goldreich and Ostrovsky [GO96] which does not leak any information at all, and the SSE construction described in [CK10] which leaks only the access and the intersection patterns.

Definition 4.3 (Leakage patterns)**.** *Let $\mathbf{q} = (q_1, \ldots, q_n)$ be a non-empty sequence of queries. For all $t \in [n]$,*

- *(access pattern) if q_t is a search query for w, then $\text{ACCP}_t(w)$ is the set $\{id(f)\}_{f \in \mathbf{f}_w}$ of identifiers of the files in $\mathbf{f}_w$ at time t.*
- *(search pattern) if q_t is a search query for w, then $\text{SP}_t(w)$ is a binary vector of length t with a 1 at location i if $q_i = q_t$ and a 0 otherwise,*
- *(addition pattern) if q_t is an add query for file f, then $\text{ADDP}_t(w)$ is a binary vector of length t with 1 at location i if $q_i = q_t$*
- *(deletion pattern) if q_t is a delete query for the identifier id, then $\text{DP}_t(w)$ is a binary vector of length t with 1 at location i if $q_i = q_t$*
- *(word pattern) $\text{WRDP}_t(w)$ is a binary vector of length t with a 1 at location i if q_i was either: (1) a search query for w; (2) an add query for a file that contained w; or (3) a delete query for a file that contained w.*

4.2 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*. 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. The latter can then use its state *st* and the proof π to verify the returned files.

To add or remove a file, the client sends a token to the provider. Note that here we assume the client will always have a local copy of any file it wishes to remove. [4] The provider updates the authenticator and returns information back to the client that will enable him to update his state *st*.

For our purposes, we need search authenticators that are not only dynamic but also *hiding* in the sense that the authenticator α reveals no information about the files.

Definition 4.4 (Dynamic search authenticator)**.** *A private-key dynamic search authenticator is a tuple of nine polynomial-time algorithms* $\mathsf{DSA} = (\mathsf{Gen}, \mathsf{Auth}, \mathsf{Chall}, \mathsf{Prove}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Add}, \mathsf{Del}, \mathsf{Update}, \mathsf{Vrfy})$ *such that:*

$K \leftarrow \mathsf{Gen}(1^k)$*: is a probabilistic algorithm that takes as input a security parameter k and outputs a secret key K.*

$(st, \alpha) \leftarrow \mathsf{Auth}(K, \delta, \mathbf{f})$*: is a probabilistic algorithm that takes as input a secret key K, an index δ and a sequence of files $\mathbf{f}$. It outputs some state st and an authenticator α.*

$c \leftarrow \mathsf{Chall}(K, w)$*: is a deterministic algorithm that takes as input a secret key K and a keyword w and outputs a challenge c.*

$\pi \leftarrow \mathsf{Prove}(\alpha, \mathbf{f}_w)$*: is a deterministic algorithm that takes as input an authenticator α and a sequence of files $\mathbf{f}_w \subseteq \mathbf{f}$ and outputs a proof π.*

$\tau_a := \mathsf{AddToken}(K, st, f)$*: is a deterministic algorithm that takes as in put a secret key K, some state information st and a file f. It outputs a token τ_a.*

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

$\tau_d := \mathsf{DelToken}(K, st, f)$*: is a deterministic algorithm that takes as in put a secret key K, some state information st and a file f. It outputs a token τ_d.*

$(\alpha', \rho) := \mathsf{Add}(\alpha, \tau_a)$*: is a deterministic algorithm that takes as input an authenticator α, a token τ_a. It outputs a new authenticator α' and a receipt ρ.*

$(\alpha', \rho) := \mathsf{Del}(\alpha, \tau_d)$*: is a deterministic algorithm that takes as input an authenticator α, a token τ_d. It outputs a new authenticator α' and a receipt ρ.*

$\{st', \perp\} \leftarrow \mathsf{Update}(st, \tau, \rho)$*: is a deterministic algorithm that takes as input a state st, a token τ and a receipt ρ and outputs either an updated state st' or $\perp$.*

$b := \mathsf{Vrfy}(K, st, w, \mathbf{f}', \pi)$*: is a deterministic algorithm that takes as input a secret key K, a state st, a keyword w, a sequence of files $\mathbf{f}' \subseteq \mathbf{f}$ and a proof π. It outputs a bit b.*

We require that search authenticators satisfy two security properties: hiding and unforgeability. Roughly speaking, the hiding property guarantees that the authenticator does not leak any useful information about the files. Unforgeability guarantees that, for any keyword w, no adversary can generate a valid proof for a set of files $\mathbf{f}' \neq \mathbf{f}_w$ – even if the adversary can choose $\mathbf{f}$ and w.

The hiding property of a DSA should guarantee that the provider cannot learn any useful information about the client's files and queries from the authenticator and the tokens. We formalize this intuition by requiring that the view of the server (i.e., the authenticator, the challenges and the add and delete tokens) generated from any adaptive query strategy be simulatable. As in the case of SSE, this exact intuition (and its formalization) is difficult to achieve efficiently so the definition must be appropriately weakened. We do this by parameterizing the hiding property with four leakage functions that capture precisely what is being leaked by the authenticator and the tokens.

Definition 4.5 (Hiding)**.** *Let* $\mathsf{DSA} = (\mathsf{Gen}, \mathsf{Auth}, \mathsf{Chall}, \mathsf{Prove}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Add}, \mathsf{Del}, \mathsf{Vrfy})$ *be dynamic private-key search authenticator and consider the following probabilistic experiments where $\mathcal{A}$ is a stateful adversary, $\mathcal{S}$ is a stateful simulator and $\mathcal{L}_1^{\mathsf{dsa}}$, $\mathcal{L}_2^{\mathsf{dsa}}$, $\mathcal{L}_3^{\mathsf{dsa}}$ and $\mathcal{L}_4^{\mathsf{dsa}}$ are leakage functions:*

$\mathbf{Real}_{\mathcal{A}}^{\mathsf{dsa}}(k)$*: the challenger begins by running $\mathsf{Gen}(1^k)$ to generate a key K. $\mathcal{A}$ outputs a pair $(\delta, \mathbf{f})$ and receives $(st, \alpha) \leftarrow \mathsf{Auth}_K(\delta, \mathbf{f})$ from the challenger. The adversary makes a polynomial number of (adaptive) challenge and update queries. For each challenge query w it receives $c \leftarrow \mathsf{Chall}_K(w)$ from the challenger. For each add and delete query (for a file f), it receives from the challenger $\tau_a \leftarrow \mathsf{AddToken}_K(st, f)$ and $\tau_d \leftarrow \mathsf{DelToken}_K(st, f)$, respectively. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.*

$\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\mathsf{dsa}}(k)$*: $\mathcal{A}$ outputs a pair $(\delta, \mathbf{f})$. Given $\mathcal{L}_1^{\mathsf{dsa}}(\delta, \mathbf{f})$, $\mathcal{S}$ generates and sends a state st and an authenticator α to the adversary. The adversary makes a polynomial number of (adaptive) challenge and update queries. For each challenge query w, the simulator is given $\mathcal{L}_2^{\mathsf{dsa}}(\delta, \mathbf{f}, w)$ and returns a challenge c to $\mathcal{A}$. For each add query f the simulator is given $\mathcal{L}_3^{\mathsf{dsa}}(\delta, \mathbf{f}, f)$ and returns a token τ_a to $\mathcal{A}$. For each delete query f the simulator is given $\mathcal{L}_4^{\mathsf{dsa}}(\delta, \mathbf{f}, f)$ and returns a token τ_d. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.*

We say that DSA *is $(\mathcal{L}_1^{\mathsf{dsa}}, \mathcal{L}_2^{\mathsf{dsa}}, \mathcal{L}_3^{\mathsf{dsa}}, \mathcal{L}_4^{\mathsf{dsa}})$-hiding if for all* PPT *adversaries $\mathcal{A}$, there exists a* PPT *simulator $\mathcal{S}$ such that*

$$\left|\Pr\left[\,\mathbf{Real}_{\mathcal{A}}^{\mathsf{dsa}}(k) = 1\,\right] - \Pr\left[\,\mathbf{Ideal}_{\mathcal{A},\mathcal{S}}^{\mathsf{dsa}}(k) = 1\,\right]\right| \leq \mathsf{negl}(k).$$

The unforgeability property of a DSA guarantees that for any keyword w, the provider cannot find a sequence of files $\mathbf{f}' \neq \mathbf{f}_w$ and a proof such that the verification algorithm accepts.

Definition 4.6 (Unforgeability)**.** *Let* $\mathsf{DSA} = (\mathsf{Gen}, \mathsf{Auth}, \mathsf{Chall}, \mathsf{Prove}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Add}, \mathsf{Del}, \mathsf{Vrfy})$ *be a dynamic private-key search authenticator and consider the following probabilistic experiment where* $\mathcal{A}$ *is a stateful adversary:*

$\mathbf{Unf}_{\mathcal{A}}(k)$*:*

1. *the challenger generates a key* $K \leftarrow \mathsf{Gen}(1^k)$*,*
2. *the adversary* $\mathcal{A}$ *outputs an index* δ *and a sequence of files* $\mathbf{f}$*,*
3. *the challenger computes* $(st, \alpha) \leftarrow \mathsf{Auth}_K(\delta, \mathbf{f})$*,*
4. *given* α *and oracle access to* $\mathsf{Chall}_K(\cdot)$*,* $\mathsf{AddToken}_K(st, \cdot)$ *and* $\mathsf{DelToken}_K(st, \cdot)$*, the adversary* $\mathcal{A}$ *outputs a keyword* w*, a sequence of receipts* $(\rho'_1, \ldots, \rho'_m)$*, a sequence of files* $\mathbf{f}' \subseteq \mathbf{f}$ *such that* $\mathbf{f}' \neq \mathbf{f}_w$ *and a proof* π'*,*
5. *the challenger computes* $b := \mathsf{Vrfy}_K(st', w, \mathbf{f}', \pi')$*, where* st' *is the result of updating* st *using* $(\rho'_1, \ldots, \rho'_m)$*,*
6. *the output of the experiment is the bit* b*,*

We say that DSA *is unforgeable if for all* PPT *adversaries* $\mathcal{A}$*,*

$$\Pr\left[\mathbf{Unf}_{\mathcal{A}}(k) = 1\right] \leq \mathsf{negl}(k).$$

4.3 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 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 views its files $\mathbf{f}$ as a sequence of fixed-length blocks $\mathbf{b}$ and tags each block in $\mathbf{b}$ (note that that number of blocks $\#\mathbf{b}$ is larger than the number of files $\#\mathbf{f}$). This results in a sequence of tags $\mathbf{t} = (t_1, \ldots, t_{\#\mathbf{b}})$ 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. The latter uses the challenge, the files and the tags to generate a short proof π that it returns to the client. Finally, the client uses the state st to verify the proof. 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.

Finally, to add a file f, the client views it as sequence of blocks $\mathbf{b}_f$ and generates a sequence of tags $\mathbf{t} = (t_1, \ldots, t_{\#\mathbf{b}_f})$ and sends the new file together with its tags 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 to the provider and to update its state locally.

Definition 4.7 (Proof of data possession)**.** *A privately-verifiable tag-based dynamic PDP is a tuple of five polynomial-time algorithms* $\mathsf{PDP} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Chall}, \mathsf{Add}, \mathsf{Del}, \mathsf{Prove}, \mathsf{Vrfy})$ *such that:*

$K \leftarrow \mathsf{Gen}(1^k, \lambda)$: *is a probabilistic algorithm that takes as input a security parameter k, a locality parameter λ and outputs a private key K.*

$(st, \mathbf{t}) \leftarrow \mathsf{Encode}(K, \mathbf{f})$: *is a probabilistic algorithm that takes as input a secret key K and a sequence of files $\mathbf{f}$ and outputs some state st and a sequence of tags $\mathbf{t}$.*

$c \leftarrow \mathsf{Chall}(K, st)$: *is a probabilistic algorithm that takes some state st as input and outputs a challenge c.*

$\pi := \mathsf{Prove}(\mathbf{f}, \mathbf{t}, c)$: *is a deterministic algorithm that takes as input a sequence of files $\mathbf{f}$, and tags $\mathbf{t}$ and a challenge c and outputs a proof π.*

$(st', \mathbf{t}') \leftarrow \mathsf{Add}(K, st, f)$: *is a deterministic algorithm that takes as input a secret key K, some state st and a file f. It outputs an updated state st' and a sequence of tags $\mathbf{t}'$.*

$st' \leftarrow \mathsf{Del}(K, st, f)$: *is a deterministic algorithm that takes as input a secret key K, some state st and a file f. It outputs an updated state st'.*

$b \leftarrow \mathsf{Vrfy}(K, st, c, \pi)$: *is a deterministic algorithm that takes as input a secret key K, a state st, a challenge c and a proof π. It output a bit b, where '$1'$ indicates acceptance and '$0'$ indicates rejection.*

A PDP is secure if it satisfies a weak soundness property which, roughly speaking, guarantees that if the client accepts the interaction, then the server did not tamper with too many of the blocks in $\mathbf{f}$. As in previous works on proofs of storage [ABC$^+$07, JK07, SW08, DVW09, AKK09] we formalize this intuition using the notion of a knowledge extractor [FFS88, BG92].

Definition 4.8 (Soundness). *Let $\mathsf{PDP} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Chall}, \mathsf{Prove}, \mathsf{Add}, \mathsf{Del}, \mathsf{Vrfy})$ be a tag-based privately-verifiable dynamic PDP and consider the following probabilistic experiment where $\mathcal{A}$ is a stateful adversary, $\mathcal{K}$ is a knowledge extractor and $\lambda \in \mathbb{N}$:*

$\mathbf{Sound}_{\mathcal{A},\mathcal{K},\lambda}(k)$:

- *the challenger generates a key $K \leftarrow \mathsf{Gen}(1^k, \lambda)$*
- *given oracle access to $\mathsf{Encode}_K(\cdot)$, the adversary $\mathcal{A}$ outputs a sequence of files $\mathbf{f}$*
- *the challenger encodes the file $(st, \mathbf{t}) \leftarrow \mathsf{Encode}_K(\mathbf{f})$*
- *given $\mathbf{t}$, $\mathcal{A}$ makes a polynomial number of queries (op, f) and for each query:*
  - *if $\mathsf{op} = \mathsf{add}$, the challenger computes $(st', \mathbf{t}_f) \leftarrow \mathsf{Add}(K, st, f)$ and adds f to $\mathbf{f}$ and $\mathbf{t}_f$ to $\mathbf{t}$*
  - *if $\mathsf{op} = \mathsf{del}$, the challenger computes $st' \leftarrow \mathsf{Del}(K, st, f)$ and updates st to st' and removes f from $\mathbf{f}$*
- *the challenger chooses at random λ indices in $[\#\mathbf{b}]$. Let Q be this set.*
- *given (st, Q, r) and oracle access to $\mathcal{A}(\mathbf{f}, \mathbf{t}, \cdot)$, the extractor $\mathcal{K}$ outputs $\mathbf{b}^*$*
- *the output of the experiment is $\mathbf{b}^*$*

We say that PDP is sound if there exists an expected polynomial-time knowledge extractor $\mathcal{K}$ such that for all PPT adversaries $\mathcal{A}$ we have:

$$\Pr\left[\mathbf{Sound}_{\mathcal{A},\mathcal{K},\lambda}(k) \neq \mathbf{b}_Q \bigwedge \langle \mathcal{A}, \mathcal{C}(1^k; r)\rangle = 1\right] \leq \varepsilon(k),$$

where $\mathbf{b}_Q$ denotes the subsequence of blocks in $\mathbf{b}$ indexed by Q and $\langle \mathcal{A}, \mathcal{C}(1^k; r)\rangle = 1$ denotes the event that the interaction of $\mathcal{A}$ with the honest client $\mathcal{C}$ using coins r is valid.

To detect with very high probability (say $1 - 2^{-k}$) a provider that modified at least an ε fraction of the blocks in $\mathbf{f}$, it suffices to execute a PDP with locality $\lambda = k/\varepsilon$.

5 The CS2 Protocol

In this Section we describe the CS2 protocol and prove its security in the ideal/real-world paradigm. A precise description of the protocol is provided in Figure 1 but, at a high level, the CS2 operation are implemented as follows:

- 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 and to authenticate δ and $\mathbf{f}$ with the search authenticator. 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 $\mathbf{t}$ 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 and the latter uses it to recover a subset $\mathbf{c}_w \subseteq \mathbf{c}$ of encrypted files that contain w. To prove to the client that $\mathbf{c}_w$ is indeed the correct set of ciphertexts, the provider generates a DSA proof π_a which it returns with $\mathbf{c}_w$. The client then verifies (on input its local state st) the proof π_a and, if correct, decrypts the ciphertexts.
- 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.

Our security definition is in the ideal/real-world paradigm which compares the real-world execution of a protocol to the ideal-world evaluation of a functionality by a trusted party [Bea92, GL91, MR92, Can00]. We refer to the ideal functionality that formally captures the properties we expect from a secure cloud storage system as the CloudStore functionality and provide a detailed description in Figure 2.

Definition 5.1 (Secure emulation). *Let Π be a two-party protocol and consider the following two probabilistic experiments where $\mathcal{C}$ is the client and $\mathcal{A}$ and $\mathcal{A}'$ are adversaries:*

$\mathbf{Real}^{\mathsf{cloud}}_{\mathcal{C},\mathcal{A},\mathbf{f}}(k)$: *at the beginning of the real-world experiment the client receives a sequence of files $\mathbf{f}$ while the adversary $\mathcal{A}$ receives only the security parameter 1^k. The outputs of the experiment consist of the outputs of $\mathcal{C}$ and $\mathcal{A}$ after executing Π for polynomially-many steps in k.*

Let $\mathsf{SSE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{SrchToken}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Search}, \mathsf{Add}, \mathsf{Del}, \mathsf{Dec})$ be an index-based private-key dynamic SSE scheme, $\mathsf{DSA} = (\mathsf{Gen}, \mathsf{Auth}, \mathsf{Prove}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Add}, \mathsf{Del}, \mathsf{Vrfy})$ be a dynamic search authenticator, $\mathsf{PDP} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Add}, \mathsf{Del}, \mathsf{Prove}, \mathsf{Vrfy})$ be a privately-verifiable PDP scheme. CS2 is defined as follows:

- $\mathsf{SETUP}_{\mathbf{C}}(1^k)$: generate keys $K_1 \leftarrow \mathsf{SSE.Gen}(1^k)$, $K_2 \leftarrow \mathsf{PDP.Gen}(1^k, k/\varepsilon)$ and $K_3 \leftarrow \mathsf{DSA.Gen}(1^k)$. Output $K = (K_1, K_2, K_3)$.
- $\mathsf{STORE}_{\mathbf{C}}(K, \mathbf{f})$: generate an index δ from $\mathbf{f}$ and compute $(\gamma, \mathbf{c}) \leftarrow \mathsf{SSE.Enc}_{K_1}(\delta, \mathbf{f})$, $(st_p, \mathbf{t}) \leftarrow \mathsf{PDP.Encode}_{K_2}(\mathbf{c})$, and $(st_a, \alpha) \leftarrow \mathsf{DSA.Auth}_{K_3}(\delta, \mathbf{c})$. Output $st = (st_p, st_a)$ and send $(\gamma, \mathbf{c}, \mathbf{t}, \alpha)$ to the server.
- $\mathsf{SEARCH}_{\mathbf{C},\mathbf{S}}\big((K, w, st), (\gamma, \mathbf{c}, \alpha)\big)$:
  1. **[Client]**: send $\tau_s \leftarrow \mathsf{SSE.SrchToken}_{K_1}(w)$ and $c_a \leftarrow \mathsf{DSA.Chall}_{K_3}(w)$
  2. **[Server]**: send $\mathbf{c}_w \leftarrow \mathsf{SSE.Search}(\gamma, \mathbf{c}, \tau)$ and $\pi_a \leftarrow \mathsf{DSA.Prove}(\alpha, c_a, \mathbf{c}_w)$
  3. **[Client]**: compute $b \leftarrow \mathsf{DSA.Vrfy}_{K_3}(st_a, \mathbf{c}_w, \pi_a)$. If $b = 0$, output $\perp$. Otherwise, for $1 \leq i \leq \#\mathbf{c}_w$, output $f_i \leftarrow \mathsf{SSE.Dec}_{K_1}(\mathbf{c}_w[i])$.
- $\mathsf{CHECK}_{\mathbf{C},\mathbf{S}}((K, st), (\mathbf{c}, \mathbf{t}))$:
  1. **[Client]**: send $c_p \leftarrow \mathsf{PDP.Chall}(st_p)$
  2. **[Server]**: send $\pi_p \leftarrow \mathsf{PDP.Prove}(\mathbf{c}, \mathbf{t}, c_p)$
  3. **[Client]**: compute $b \leftarrow \mathsf{PDP.Vrfy}_{K_2}(st_p, c_p, \pi_p)$. If $b = 1$ output 1 otherwise output $\perp$.
- $\mathsf{ADD}_{\mathbf{C},\mathbf{S}}((K, st, f), (\gamma, \mathbf{c}, \alpha))$:
  1. **[Client]**: send $\tau_a \leftarrow \mathsf{SSE.AddToken}_{K_1}(f)$ and $\tau_a' \leftarrow \mathsf{DSA.AddToken}_K(st_a, f)$
  2. **[Server]**: compute $(\gamma', \mathbf{c}') \leftarrow \mathsf{SSE.Add}(\gamma, \mathbf{c}, \tau_a)$ and $(\alpha', \rho) \leftarrow \mathsf{DSA.Add}(\alpha, \tau_a')$ and send ρ. Update α to α', γ to γ' and $\mathbf{c}$ to $\mathbf{c}'$
  3. **[Client]**: compute $u \leftarrow \mathsf{DSA.Update}(st_a, \rho)$ and update st_a to u if $u \neq \perp$. Output $\perp$ and halt otherwise.
- $\mathsf{DELETE}_{\mathbf{C},\mathbf{S}}((K, st, f), (\gamma, \mathbf{c}, \alpha))$:
  1. **[Client]**: send $\tau_d \leftarrow \mathsf{SSE.DelToken}_{K_1}(id(f))$ and $\tau_d' \leftarrow \mathsf{DSA.AddToken}_K(st_a, f)$
  2. **[Server]**: compute $(\gamma', \mathbf{c}') \leftarrow \mathsf{SSE.Del}(\gamma, \mathbf{c}, \tau_d)$ and $(\alpha', \rho) \leftarrow \mathsf{DSA.Del}(\alpha, \tau_d')$ and send ρ. Update α to α', γ to γ' and $\mathbf{c}$ to $\mathbf{c}'$
  3. **[Client]**: compute $u \leftarrow \mathsf{DSA.Update}(st_a, \rho)$ and update st_a to u if $u \neq \perp$. Output $\perp$ and halt otherwise.

Figure 1: The CS2 protocol.

The CloudStore functionality is parameterized with leakage functions $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$ and $\mathcal{L}_4$ and soundness parameter $\varepsilon \in [0,1]$. It provides the client access to five operations $\mathsf{STORE}, \mathsf{SEARCH}, \mathsf{CHECK}, \mathsf{ADD}, \mathsf{DELETE}$ defined as follows:

- $\mathsf{STORE}(\delta, \mathbf{f})$: store $(\delta, \mathbf{f})$ and send $\mathcal{L}_1(\delta, \mathbf{f})$ to $\mathcal{A}$.
- $\mathsf{SEARCH}(w)$: send $\mathcal{L}_2(\delta, \mathbf{f}, w)$ to $\mathcal{A}$ and ask if it wants to return an incorrect set of files. If so return $\perp$ and halt, otherwise return $\mathbf{f}_w$ to the client.
- $\mathsf{CHECK}(\cdot)$: ask $\mathcal{A}$ if it wishes to tamper with at least an ε fraction of $\mathbf{f}$. If so, return 0 to the client otherwise return 1.
- $\mathsf{ADD}(f)$: send $\mathcal{L}_3(\delta, \mathbf{f}, f)$ to $\mathcal{A}$ and ask it if it wishes to abort. If so return $\perp$ to the client and halt, otherwise store f.
- $\mathsf{DELETE}(f)$: send $\mathcal{L}_4(\delta, \mathbf{f}, f)$ to $\mathcal{A}$ and ask it if it wishes to abort. If so return $\perp$ to the client and halt, otherwise delete f.

Figure 2: The CloudStore functionality.

$\mathbf{Ideal}^{\mathsf{cloud}}_{\mathcal{C},\mathcal{A}',\mathbf{f}}(k)$: *at the beginning of the ideal-world experiment the client receives a sequence of files* $\mathbf{f}$ *while the ideal adversary* $\mathcal{A}'$ *receives only the security parameter* 1^k. *The outputs of the experiment consists of the outputs of* $\mathcal{C}$ *and* $\mathcal{A}'$ *after interacting with the* CloudStore *functionality for polynomially-many steps in* k.

We say that Π *securely emulates the* CloudStore *functionality if for all* PPT *adversaries* $\mathcal{A}$, *there exists an ideal expected polynomial-time adversary* $\mathcal{A}'$ *such that*

$$\left\{ \mathbf{Real}^{\mathsf{cloud}}_{\mathcal{C},\mathcal{A},\mathbf{f}}(k) \right\}_{k,\mathbf{f}} \overset{c}{\approx} \left\{ \mathbf{Ideal}^{\mathsf{cloud}}_{\mathcal{C},\mathcal{A}',\mathbf{f}}(k) \right\}_{k,\mathbf{f}}.$$

We now show that the CS2 protocol securely emulates CloudStore.

Theorem 5.2. *If* SSE *is* $(\mathcal{L}^{\mathsf{sse}}_1, \mathcal{L}^{\mathsf{sse}}_2, \mathcal{L}^{\mathsf{sse}}_3, \mathcal{L}^{\mathsf{sse}}_4)$*-secure against adaptive chosen keyword attacks, if* DSA *is unforgeable and* $(\mathcal{L}^{\mathsf{dsa}}_1, \mathcal{L}^{\mathsf{dsa}}_2, \mathcal{L}^{\mathsf{dsa}}_3, \mathcal{L}^{\mathsf{dsa}}_4)$*-hiding and if* PDP *is* ε*-sound then the CS2 protocol securely emulates the* CloudStore *functionality with soundness* ε *and leakage* $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$ *where* $\mathcal{L}_i = (\mathcal{L}^{\mathsf{sse}}_i, \mathcal{L}^{\mathsf{dsa}}_i)$ *for* $1 \leq i \leq 4$.

Proof. Let $\mathcal{S}_{\mathsf{SSE}}$ and $\mathcal{S}_{\mathsf{DSA}}$ be the simulators guaranteed to exist by the CKA2-security and the hiding property of SSE and DSA, respectively. Let $\mathcal{K}$ be the knowledge extractor guaranteed to exist by the soundness of PDP. Consider the ideal adversary $\mathcal{A}'$ that simulates $\mathcal{A}$ and proceeds as follows when receiving messages from the CloudStore functionality:

- (STORE) upon receiving $\mathcal{L}_1(\delta, \mathbf{f}) = (\mathcal{L}^{\mathsf{sse}}_1(\delta, \mathbf{f}), \mathcal{L}^{\mathsf{dsa}}_1(\delta, \mathbf{f}))$ from CloudStore, it computes $(\gamma, \mathbf{c}) \leftarrow \mathcal{S}_{\mathsf{SSE}}(\mathcal{L}^{\mathsf{sse}}_1(\delta, \mathbf{f}))$ and $(st_a, \alpha) \leftarrow \mathcal{S}_{\mathsf{DSA}}(\mathcal{L}^{\mathsf{dsa}}_1(\delta, \mathbf{f}))$. It then generates $K_2 \leftarrow \mathsf{PDP.Gen}(1^k)$ and computes $(st_p, \mathbf{t}) \leftarrow \mathsf{PDP.Encode}_{K_2}(\mathbf{c})$. It sends $(\gamma, \mathbf{c}, \mathbf{t}, \alpha)$ to $\mathcal{A}$.
- (SEARCH) upon receiving $\mathcal{L}_2(\delta, \mathbf{f}, w) = (\mathcal{L}^{\mathsf{sse}}_2(\delta, \mathbf{f}, w), \mathcal{L}^{\mathsf{dsa}}_2(\delta, \mathbf{f}, w))$ from CloudStore, it sends $\tau_s \leftarrow \mathcal{S}_{\mathsf{SSE}}(\mathcal{L}_2(\delta, \mathbf{f}, w))$ and $c_a \leftarrow \mathcal{S}_{\mathsf{DSA}}(\mathcal{L}^{\mathsf{dsa}}_2(\delta, \mathbf{f}, w))$ to $\mathcal{A}$. The latter returns a sequence of ciphertexts $\mathbf{c}^*$ and a proof π^*_a. When asked by CloudStore if it wants to return an incorrect set of files, $\mathcal{A}'$ computes $\mathbf{c}_w \leftarrow \mathsf{SSE.Search}(\gamma, \mathbf{c}, \tau_s)$ and responds NO if $\mathbf{c}^* = \mathbf{c}_w$ and YES otherwise.
- (CHECK) When asked by CloudStore if it wants to tamper with any file, $\mathcal{A}'$ chooses a random subset Q of k/ε blocks of $\mathbf{c}$ (where $\mathbf{b}$ denotes the "block view" of $\mathbf{c}$) and runs $\mathcal{K}$ with oracle access to $\mathcal{A}$. It answers YES if $\mathcal{K}$ returns $\mathbf{b}^* \neq \mathbf{b}_Q$ and NO otherwise.

- (ADD) upon receiving $\mathcal{L}_3(\delta, \mathbf{f}, f) = (\mathcal{L}_3^{\mathsf{sse}}(\delta, \mathbf{f}, f), \mathcal{L}_3^{\mathsf{dsa}}(\delta, \mathbf{f}, f))$ from CloudStore, it sends $\tau_a \leftarrow \mathcal{S}_{\mathsf{SSE}}(\mathcal{L}_3^{\mathsf{sse}}(\delta, \mathbf{f}, f))$ and $\tau_a' \leftarrow \mathcal{S}_{\mathsf{DSA}}(\mathcal{L}_3^{\mathsf{dsa}}(\delta, \mathbf{f}, f))$ to $\mathcal{A}$. The adversary then returns ρ^* which the simulator uses to compute $u \leftarrow \mathsf{DSA.Update}(st_a, \tau_a', \rho^*)$. If $u = \perp$, the simulator answers CloudStore with YES, otherwise it answers NO.

- (DELETE) upon receiving $\mathcal{L}_4(\delta, \mathbf{f}, f) = (\mathcal{L}_4^{\mathsf{sse}}(\delta, \mathbf{f}, f), \mathcal{L}_4^{\mathsf{dsa}}(\delta, \mathbf{f}, f))$ from CloudStore, it send $\tau_d \leftarrow \mathcal{S}_{\mathsf{SSE}}(\mathcal{L}_4^{\mathsf{sse}}(\delta, \mathbf{f}, f))$ and $\tau_d' \leftarrow \mathcal{S}_{\mathsf{DSA}}(\mathcal{L}_4^{\mathsf{dsa}}(\delta, \mathbf{f}, f))$ to $\mathcal{A}$. The adversary then returns ρ^* which the simulator uses to compute $u \leftarrow \mathsf{DSA.Update}(st_a, \tau_d', \rho^*)$. If $u = \perp$, the simulator answers CloudStore with YES, otherwise it answers NO.

The indistinguishability of $(\gamma, \mathbf{c}, \mathbf{t}, \alpha)$ during STORE operations follow from the CKA2-security of SSE, the CPA-security of SKE and the hiding property of DSA. During SEARCH operations, the indistinguishability of τ_s and c_a follow from the CKA2-security of SSE and the hiding property of DSA. In addition, the unforgeability of DSA implies that, with all but negligible probability in k, the answer $\mathcal{A}'$ returns to CloudStore (when asked whether it wishes to return the correct files) is distributed correctly (i.e., according to what $\mathcal{A}$ would answer in a $\mathbf{Real}^{\mathsf{cloud}}_{\mathcal{C},\mathcal{A},\mathbf{f}}(k)$ experiment). Similarly, the ε-soundness of PDP and the hiding and unforgeability properties of DSA guarantee that the answers returned by $\mathcal{A}'$ during the CHECK and ADD/DELETE operations are distributed correctly. □

6 Explicit Constructions

6.1 A Dynamic Searchable Symmetric Encryption Scheme

Our scheme is an extension of the SSE-1 construction from [CGKO06, CGKO11] 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 [CGKO06, CGKO11]). 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 $\mathtt{A}_s$. The 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 size at least $|\mathbf{c}|/8$ and 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 $\mathbf{f}$ [5]. It is important to note, however, that random storage by itself is not enough to hide the length of the lists since the encryption

[5]Of course, the lists could be stored at fixed locations if they were all padded to the same length but this would be wasteful in terms of storage.

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 $\mathcal{L}_w$, 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 $\mathsf{SSE\text{-}1}$ dynamic. As mentioned above, the limitations of $\mathsf{SSE\text{-}1}$ are twofold: (1) it is only CKA1-secure and (2) it is not explicitly dynamic. As observed in [CK10], 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 as follows:

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 recover pointers to the nodes that correspond to the file being deleted. 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 ever 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 location in $\mathtt{A}_s$.

Our construction is described in detail in Figures 3, 4 and 5. In the following we show that it is CKA2-secure in the random oracle model with respect to the following leakage functions:

1. $\mathcal{L}_1^{\mathsf{sse}}(\delta, \mathbf{f}) = (\#\mathtt{A}, \#f, \#W, |f_1, \ldots, |f_{\#\mathbf{f}}|)$,

2. $\mathcal{L}_2^{\mathsf{sse}}(\delta, \mathbf{f}, w) = (\textsc{accp}(w), \textsc{sp}(w), \textsc{wrdp}(w))$,

3. $\mathcal{L}_3^{\mathsf{sse}}(\delta, \mathbf{f}, f) = (\textsc{addp}(w), \textsc{wrdp}(w_1), \ldots, \textsc{wrdp}(w_{\#f}), \#f, |f|)$,

4. $\mathcal{L}_4^{\mathsf{sse}}(\delta, \mathbf{f}, f) = (\textsc{dp}(w), \#f, id(f))$.

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:
     $$\mathtt{N}_i := \Big(\langle id_i, \mathsf{addr}_s(\mathtt{N}_{i-1}), \mathsf{addr}_s(\mathtt{N}_{i+1}) \rangle \oplus H_1(K_w, r_i), r_i \Big)$$
     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:
     $$\mathtt{D}_i := \Big(\langle \mathsf{addr}_d(\mathtt{D}_{i+1}), \mathsf{addr}_d(\mathtt{N}^\star_{j-1}), \mathsf{addr}_d(\mathtt{N}^\star_{j+1}), \mathsf{addr}_s(\mathtt{N}_j), \mathsf{addr}_s(\mathtt{N}_{j-1}), \mathsf{addr}_s(\mathtt{N}_{j+1}), F_{K_1}(w) \rangle \oplus H_1(K_f, r_i'), r_i' \Big)$$
     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 $(\mathtt{F}_1, \ldots, \mathtt{F}_z)$ and $(\mathtt{F}'_1, \ldots, \mathtt{F}'_z)$ be the free nodes in $\mathtt{A}_s$ and $\mathtt{A}_d$, respectively. Set
     $$\mathtt{T}_s[F_{K_1}(\mathsf{free})] := \langle \mathsf{addr}_s(\mathtt{F}_z), \mathbf{0}^{\log \#\mathtt{A}} \rangle \oplus G_{K_2}(\mathsf{free})$$
     and for $z \le i \le 1$, set
     $$\mathtt{A}_s[\mathsf{addr}_s(\mathtt{F}_i)] := \langle \mathbf{0}^{\log \#\mathbf{f}}, \mathsf{addr}_s(\mathtt{F}_{i-1}), \mathsf{addr}_d(\mathtt{F}_i^\star) \rangle \oplus Q_{K_4}(i)$$
     where $\mathsf{addr}_s(\mathtt{F}_0) = \mathbf{0}^{\log \#\mathtt{A}}$.
  5. fill the remaining entries of $\mathtt{A}_s$ and $\mathtt{A}_d$ with random strings
  6. for $1 \le i \le \#\mathbf{f}$, 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 $\mathbf{c} = (c_1, \ldots, c_{\#\mathbf{f}})$.

Figure 3: A fully dynamic SSE scheme (Part 1).

- $\mathsf{SrchToken}(K, w)$: 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 $1 \leq i \leq \#f$:

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

  and r_i is a random k-bit string. Update $\#\mathsf{L}_{\mathsf{free}}$ to $\#\mathsf{L}_{\mathsf{free}} - \#f$.
- $\mathsf{Add}(\gamma, \mathbf{c}, \tau_a)$:
  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:

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

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

Figure 4: A fully dynamic SSE scheme (Part 2).

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

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

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

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

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

Figure 5: A fully dynamic SSE scheme (Part 3).

Theorem 6.1. *If* SKE *is CPA-secure and if* F, G, P *and* Q *are pseudo-random, then* SSE *as described in Figures 3, 4 and 5 is* $(\mathcal{L}_1^{\mathsf{sse}}, \mathcal{L}_2^{\mathsf{sse}}, \mathcal{L}_3^{\mathsf{sse}}, \mathcal{L}_4^{\mathsf{sse}})$*-secure against adaptive chosen-keyword attacks in the random oracle model.*

Proof Sketch. We describe a polynomial-time simulator $\mathcal{S}$ such that for all probabilistic polynomial-time adversaries $\mathcal{A}$, the outputs of $\mathbf{Real}_{\mathcal{A}}^{\mathsf{sse}}(k)$ and $\mathbf{Ideal}_{\mathcal{S}}^{\mathsf{sse}}(k)$ are computationally indistinguishable. Consider the simulator $\mathcal{S}$ that adaptively simulates an encrypted index $\widetilde{\gamma} = (\widetilde{\mathtt{A}}_s, \widetilde{\mathtt{T}}_s, \widetilde{\mathtt{A}}_d, \widetilde{\mathtt{T}}_d)$, a sequence of simulated ciphertexts $\widetilde{\mathbf{c}}$ and $n \in \mathbb{N}$ simulated tokens $(\widetilde{\tau}_1, \ldots, \widetilde{\tau}_n)$ as follows:

- given $\mathcal{L}_1(\delta, \mathbf{f}) = (\#\mathtt{A}, \#f, \#W, |f_1|, \ldots, |f_{\#\mathbf{f}}|)$, generate $K_5 \leftarrow \mathsf{SKE.Gen}(1^k)$,
- (Simulating $\mathtt{A}_s$) let $\widetilde{\mathtt{A}}_s$ be an array of size $|\mathbf{c}|/8 + z$:
  1. fill $\widetilde{\mathtt{A}}_s$ with random strings of the form $\langle \widetilde{\mathtt{N}}, \widetilde{r} \rangle$ such that $|\widetilde{\mathtt{N}}| = 2 \log \#\mathtt{A} + \log n$ and $|\widetilde{r}| = k$,
  2. keep the following "meta-data" for each cell in $\widetilde{\mathtt{A}}_s$: its address, a file ID initialized to 0, a bit indicating whether it is open or closed (initialized to closed), a previous and a next pointer both initialized to 0, the ciphertext $\widetilde{\mathtt{N}}$ and the randomness $\widetilde{r}$ stored in the cell, a keyword initialized to $\perp$, a label initialized to $\perp$, the address of its dual (initialized to a randomly chosen address in $\widetilde{\mathtt{A}}_d$) and a bit indicating whether it is a free entry (initialized to 0).
  3. choose z cells at random and set their free bit to 1. Set $\#\mathcal{L}_{\mathsf{free}} = z$ and initialize a lookup table $\mathtt{Q}$.
- (Simulating $\mathtt{T}_s$) let $\widetilde{\mathtt{T}}_s$ be a lookup table of size $\#W$:
  1. for $1 \leq i \leq \#W + 1$ store a random $(2 \log \#\mathtt{A})$-bit string $\widetilde{v}$ in $\widetilde{\mathtt{T}}_s$ under a random k-bit search key $\widetilde{\sigma}$,
  2. keep the following meta-data for each entry in $\widetilde{\mathtt{T}}_s$: a bit indicating whether it is open or closed (initialized to closed), a head and a dual pointer both initialized to 0, a search key initialized to $\widetilde{\sigma}$, a ciphertext initialized to $\widetilde{v}$, a label initialized to $\perp$ and a bit indicating whether it is the free entry,
  3. choose one of the entries at random and set its free entry bit to 1. In the following we will denote its search key $\widetilde{\sigma}_{\mathsf{free}}$ and its ciphertext $\widetilde{v}_{\mathsf{free}}$.
- (Simulating $\mathtt{A}_d$) let $\widetilde{\mathtt{A}}_d$ be an array of size $|\mathbf{c}|/8 + z$:
  1. fill $\widetilde{\mathtt{A}}_d$ with random strings of the form $\langle \widetilde{\mathtt{N}}, \widetilde{r} \rangle$ such that $|\widetilde{\mathtt{N}}| = 6 \log \#\mathtt{A} + k$ and $|\widetilde{r}| = k$,
  2. keep the following meta-data for each cell in $\widetilde{\mathtt{A}}_d$: its address, a bit indicating whether it is open or closed (initialized to closed), a next pointer initialized to 0, the ciphertext $\widetilde{\mathtt{N}}$ and randomness $\widetilde{r}$ stored in the cell, a label initialized to $\perp$, the address of its dual, and a bit indicating whether it is free (initialized to 0).
- (Simulating $\mathtt{T}_d$) let $\widetilde{\mathtt{T}}_d$ be a lookup table of size $\#\mathbf{f}$:
  1. for $1 \leq i \leq n$ store a random $(\log \#\mathtt{A})$-bit string $\widetilde{v}$ in $\widetilde{\mathtt{T}}_d$ under a random k-bit search key $\widetilde{\sigma}$.
- (Simulating ciphertexts) for $1 \leq i \leq \#\mathbf{f}$, let $\widetilde{c}_i \leftarrow \mathsf{SKE.Enc}_{K_5}(0^{|f_i|})$.
- (Simulating search tokens) given $\mathcal{L}_2(\delta, \mathbf{f}, w) = (\textsc{accp}_t(w), \textsc{sp}_t(w), \textsc{wrdp}_t(w))$, for the t^{th} query check the search pattern and let t^- be the last time this query was made. Simulate the token as follows:

1. if $t^- < t$:
   (a) if $\#\textsc{accp}_{t^-}(w) < \#\textsc{accp}_t(w)$, sample a sequence Γ_1 of $\#\{\textsc{accp}_t(w) \setminus \textsc{accp}_{t^-}(w)\}$ cells uniformly at random from the cells in $\widetilde{\mathtt{A}}_s$ that are closed and that are not free. Set their labels to be the same as the cells whose file identifiers are in $\textsc{accp}_{t^-}(w)$ (note that these labels will all be the same), their file identifiers to the identifiers in $\textsc{accp}_{t^-}(w)\setminus\textsc{accp}_t(w)$ and set their previous and next pointers according to Γ_1. Finally, mark these cells as open.
   (b) output the token that was returned at time t^-.
2. if $t^- = t$:
   (a) sample a sequence Γ_2 of $\#\textsc{accp}_t(w)$ cells uniformly at random from the cells in $\widetilde{\mathtt{A}}_s$ that are not free. Set their labels to a randomly chosen k-bit string K (note that here all the cells should have the *same* random label K) and their previous and next pointers according to Γ_2. Finally, mark these cells as open.
   (b) check the word pattern and if this keyword has appeared previously (i.e., during the addition or deletion of a file), let $\mathtt{E}$ be the entry in $\widetilde{\mathtt{T}}_s$ that was used to simulate the token at that time. If not, choose an entry in $\widetilde{\mathtt{T}}_s$ at random that is not the free entry and denote it $\mathtt{E}$. Set $\mathtt{E}$'s label to K and output $\widetilde{\tau}_t = (K, \widetilde{\sigma}_{\mathtt{E}}, \widetilde{v}_{\mathtt{E}} \oplus \langle \alpha_1, \alpha_1\star\rangle)$ where $\widetilde{\sigma}_{\mathtt{E}}$ and $\widetilde{v}_{\mathtt{E}}$ are $\mathtt{E}$'s search key and ciphertext, respectively, α_1 is the address of the first cell of Γ_2 and $\alpha_1^\star$ is its dual.

- (Simulating add tokens) given $\mathcal{L}_3(\delta, \mathbf{f}, f) = (\textsc{addp}_t(f), \textsc{wrdp}_t(w_1), \ldots, \textsc{wrdp}_t(w_{\#f}), \#f, |f|)$ for the t^{th} query, check the addition pattern and let t^- be the last time this file was added:

  1. if $t^- < t$, let $\widetilde{\tau}_{t^-} = (\widetilde{\tau}_1, \ldots, \widetilde{\tau}_4, \widetilde{\lambda}_1, \ldots, \widetilde{\lambda}_{\#f}, \widetilde{c})$ be the token that was returned at time t^-. Modify $\widetilde{\tau}_{t^-}$ as follows: for $1 \le i \le \#f$, if $\#\mathsf{L}_{\mathsf{free}}+1-i$ is not in $\mathtt{Q}$ then set $\lambda_i[3]$ to be a random $(2\log\#\mathtt{A})$-bit string and add this string to $\mathtt{Q}$ under search key $\#\mathsf{L}_{\mathsf{free}}+1-i$; otherwise, set it to the string stored in $\mathtt{Q}$ under key $\#\mathsf{L}_{\mathsf{free}}+1-i$. Output the modifed token as $\widetilde{\tau}_t$
  2. if $t^- = t$, output $\widetilde{\tau}_t := (K, \widetilde{\sigma}_{\mathsf{free}}, \widetilde{v}_{\mathsf{free}}, \#f, \widetilde{\lambda}_1, \ldots, \widetilde{\lambda}_{\#f}, \widetilde{c})$ such that:
     (a) K is a random k-bit string if the file has never been deleted (this can be verified using the deletion pattern). Otherwise, set K to be the same K used in step $2(d)$ of the delete token simulation at that time.
     (b) check the word pattern and if the i^{th} word of f has appeared previously (during the addition or deletion of a file), let $\mathtt{E}$ be the entry in $\widetilde{\mathtt{T}}_s$ that was used to simulate that token at that time. If not, choose a random entry in $\widetilde{\mathtt{T}}_s$ that is not free and denote it $\mathtt{E}$. If $\#\mathsf{L}_{\mathsf{free}}+1-i$ is not in $\mathtt{Q}$ then let $\widetilde{Q}$ be a random $(2\log\#\mathtt{A})$-bit string and add it to $\mathtt{Q}$ under search key $\#\mathsf{L}_{\mathsf{free}}+1-i$, otherwise set $\widetilde{Q}$ to the string stored in $\mathtt{Q}$ under search key $\#\mathsf{L}_{\mathsf{free}}+1-i$. Set $\widetilde{\lambda}_i = (\widetilde{\sigma}_{\mathtt{E}}, \widetilde{v}_{\mathtt{E}}, \widetilde{Q}, s)$, where $\widetilde{\sigma}_{\mathtt{E}}$ and $\widetilde{v}_{\mathtt{E}}$ $\mathtt{E}$'s address and ciphertext, rspectively, and s is a random $(\log\#\mathbf{f} + 2\log\#\mathtt{A})$-bit string.
  3. $\widetilde{c} \leftarrow \mathsf{SKE.Enc}_{K_5}(0^{|f|})$.
  4. set $\#\mathsf{L}_{\mathsf{free}}$ to $\#\mathsf{L}_{\mathsf{free}} - \#f$

- (Simulating delete tokens) given $\mathcal{L}_4(\delta, \mathbf{f}, f) = (\textsc{dp}_t(f), \#f, id(f))$, for the t^{th} query, check the deletion pattern and let t^- be the last time this file was deleted:

  1. if $t^- < t$, let $\widetilde{\tau}_{t^-} = (\widetilde{\tau}_1, \ldots, \widetilde{\tau}_{5+\#f}, \#f, id)$ be the token that was returned at time t^-. Modify $\widetilde{\tau}_{t^-}$ as follows: for $1 \le i \le \#f$, if $\#\mathsf{L}_{\mathsf{free}}+i$ is not in $\mathtt{Q}$ then set τ_{5+i} to be a random $(2\log\#\mathtt{A})$-bit string and add this string to $\mathtt{Q}$ under search key $\#\mathsf{L}_{\mathsf{free}}+i$; otherwise set it to the string stored in $\mathtt{Q}$ under key $\#\mathsf{L}_{\mathsf{free}}+i$. Output the modifed token as $\widetilde{\tau}_t$

2. if $t^{-} = t$,
   (a) find the cells in $\widetilde{\mathtt{A}}_s$ that are open and that have $id(f)$ as file identifier. Let ν be the total number of these cells and note that $\nu \leq \#f$,
   (b) choose $\#f - \nu$ closed cells in $\mathtt{A}_s$ at random,
   (c) set to 1 the free bit of all the cells chosen in steps $2(a)$ and $2(b)$,
   (d) let Γ_3 be the duals of the cells chosen in steps $2(a)$ and $2(b)$ permuted at random. Set their labels to a randomly chosen k-bit string K and their next and previous pointers according to Γ_3,
   (e) for $1 \leq i \leq \#f$, if $\#\mathsf{L}_{\mathsf{free}} + i$ is not in $\mathtt{Q}$ then let $\widetilde{Q}_i$ be a random $(2\log\#\mathcal{A})$-bit string; otherwise set it to be the string in $\mathtt{Q}$ stored under search key $\#\mathsf{L}_{\mathsf{free}} + i$.
   (f) mark the free entry in $\widetilde{\mathtt{T}}_s$ as open
   (g) set $\widetilde{\tau}_t = (K, \widetilde{\sigma}, \alpha_1 \oplus \widetilde{v}, \widetilde{\sigma}_{\mathsf{free}}, \widetilde{v}_{\mathsf{free}}, \widetilde{Q}_1, \ldots, \widetilde{Q}_{\#f}, \#f, id(f))$, where $\widetilde{\sigma}$ is the address of a randomly chosen closed entry in $\widetilde{\mathtt{T}}_d$ and $\widetilde{v}$ is its ciphertext, and α_1 is the address of the first cell in Γ_3.
3. set $\#\mathsf{L}_{\mathsf{free}}$ to $\#\mathsf{L}_{\mathsf{free}} + \#f$

- (Answering H_1 queries) given query (K, r), check if the query was made before. If so return whatever was returned previously. If not, find a cell in $\widetilde{\mathtt{A}}_s$ with label K and randomness r. If such a cell exists return $\langle id, \alpha_{-1}, \alpha_{+1}\rangle \oplus v$, where id, α_{-1}, α_{+1} and v are the identifiers, previous pointer, next pointer and ciphertext of the cell. If such a cell does not exist return and store a random $(\log\#\mathbf{f} + 2\log\#\mathtt{A})$-bit string.

- (Answering H_2 queries) given query (K, r), check if the query was made before. If so return whatever was returned previously. If not, search for a cell in $\widetilde{\mathtt{A}}_d$ with label K and randomness r. If such a cell exists, denote it $\mathtt{C}$ and return $\langle \alpha_1, \ldots, \alpha_6, \mu\rangle \oplus \widetilde{v}_{\mathtt{C}}$, where α_1 is the address of a randomly chosen cell in $\mathtt{A}_d$ that is closed and that has label K, α_2 is the address of the dual of the node that precedes the dual of $\mathtt{C}$, α_3 is the address of the dual of the node that follows the dual of $\mathtt{C}$, α_4 is the address of the dual of $\mathtt{C}$, α_5 is the address of the node that precedes the dual of $\mathtt{C}$ and α_6 is the address of the node that follows the dual of $\mathtt{C}$ and μ is a random k-bit string if $\mathtt{C}$'s dual is closed and is the address of the entry in $\widetilde{T}_s$ with label K otherwise.

$\widetilde{\mathtt{A}}_s$ and $\widetilde{\mathtt{A}}_d$ are distributed identically to $\mathtt{A}_s$ and $\mathtt{A}_d$. The indistinguishability of $\widetilde{\mathtt{T}}_s$ and $\widetilde{\mathtt{T}}_d$ from $\mathtt{T}_s$ and $\mathtt{T}_d$, respectively, follows from the pseudo-randomness of G. The indistinguishability of $\widetilde{\tau}_s$ follows from the pseudo-randomness of F, G and P and that of $\widetilde{\tau}_a$ and $\widetilde{\tau}_d$ from the pseudo-randomness of F, G, P and Q. Finally, the indistinguishability of $\widetilde{\mathbf{c}}$ follows from the CPA-security of SKE. $\square$

6.2 A Dynamic Search Authenticator Scheme

In this section we describe our dynamic search authenticator. The construction is described in Figure 6. We show its security in the following Theorem.

Theorem 6.2. *If G is pseudo-random and if* H *and* IH *are collision-resistant, then* DSA *is unforgeable.*

Proof Sketch. We show that if there exists a PPT adversary $\mathcal{A}$ that succeeds in the $\mathbf{Unf}_{\mathcal{A}}(k)$ experiment with non-negligible probability then there exists a PPT adversary $\mathcal{B}$ that breaks the collision-resistance of either H, IH or G (note that collision-resistance is implied by pseudo-randomness).

$\mathcal{B}$ begins by simulating $\mathcal{A}$. It generates keys for F and G and uses them to answer $\mathcal{A}$'s oracle queries. Recall that $\mathcal{A}$ outputs a sequence of receipts $(\rho'_1, \ldots, \rho'_m)$, a set of files $\mathbf{f}'$ and a proof π'. $\mathcal{B}$ uses the receipts to update its state and aborts if at any point the update algorithm outputs $\perp$. Conditioned

Let $F : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^*$ and $G : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^*$ be a pseudo-random functions. Let $\mathrm{H} : \{0,1\}^* \to \{0,1\}^k$ be a hash function and let $\mathrm{IH} : \{0,1\}^* \to \{0,1\}^k$ be an incremental hash function. Construct a dynamic search authenticator scheme $\mathsf{DSA} = (\mathsf{Gen}, \mathsf{Auth}, \mathsf{Chall}, \mathsf{Prove}, \mathsf{AddToken}, \mathsf{DelToken}, \mathsf{Add}, \mathsf{Del}, \mathsf{Update}, \mathsf{Vrfy})$ as follows:

- $\mathsf{Gen}(k)$: samples two k-bit keys K_1 and K_2 uniformly at random and output $K = (K_1, K_2)$
- $\mathsf{Auth}(K, \delta, \mathbf{f})$: for all $w \in W$ let $\lambda_w := \langle F_{K_1}(w), \sum_{f \in \mathbf{f}_w} \mathrm{IH}(G_{K_2}(f)) \rangle$. Construct a Merkle tree $\mathtt{MT}$ with leaves $\{\lambda_w : w \in W\}$ permuted in a random order and output (st, α), where st is the root of the tree and α is the tree itself.
- $\mathsf{Chall}(K, w)$: output $c := F_{K_1}(w)$.
- $\mathsf{Prove}(\alpha, c)$: find the leaf λ_w in α whose first element is c and output the path from λ_w to the root as the proof π
- $\mathsf{AddToken}(K, st, f)$: parse f as $(w_1, \ldots, w_n)$ and output $\tau_a := (F_{K_1}(w_1), \ldots, F_{K_1}(w_n), G_{K_2}(f))$.
- $\mathsf{DelToken}(K, st, f)$: parse f as $(w_1, \ldots, w_n)$ and output $\tau_d := (F_{K_1}(w_1), \ldots, F_{K_1}(w_n), G_{K_2}(f))$.
- $\mathsf{Add}(\alpha, \tau_a)$:

```
1. parse $\tau$ as $(\tau_1, \ldots, \tau_n, h)$
2. for $1 \le i \le n$,
   (a) find the leaf $\lambda_i$ in $\alpha$ whose first element is $\tau_i$
   (b) parse $\lambda_i$ as $(\lambda_{i,1}, \lambda_{i,2})$ and let $\lambda'_i := (\lambda_{i,1}, \lambda_{i,2} + \mathrm{IH}(h))$
   (c) let $\rho_i$ be the path in $\mathtt{MT}$ from $\lambda_i$ to the root
3. let $\mathtt{MT}'$ is the Merkle tree with leaves $\lambda_1, \ldots, \lambda_n$ replaced with $\lambda'_1, \ldots, \lambda'_n$
4. output $(\alpha', \rho)$ where $\alpha' = \mathtt{MT}'$ and $\rho = (\rho_1, \ldots, \rho_n)$
```

- $\mathsf{Del}(\alpha, \tau_d)$: is the same as Add with the exception that $\lambda'_i := (\lambda_{i,1}, \lambda_{i,2} - \mathrm{IH}(h))$,
- $\mathsf{Update}(st, \tau, \rho)$:

```
1. parse $\tau$ as $(\tau_1, \ldots, \tau_{n+1})$
2. parse $\rho$ as $(\rho_1, \ldots, \rho_n)$ and let $\lambda_i = \langle \lambda_{i,1}, \lambda_{i,2} \rangle$ be the leaf of $\rho_i$
3. for $1 \le i \le n$, if the Merkle tree verification of $\rho_i$ using $st$ fails output $\perp$
4. for $1 \le i \le n$, if $\tau$ is an add token compute $\lambda'_i := \langle \lambda_{i,1}, \lambda_{i,2} + \tau_{n+1} \rangle$; otherwise compute
   $\lambda'_i := \langle \lambda_{i,1}, \lambda_{i,2} - \tau_{n+1} \rangle$
5. update the root hash using $(\lambda'_1, \ldots, \lambda'_n)$ and output it as $st'$
```

- $\mathsf{Vrfy}(K, st, w, \mathbf{f}', \pi)$: compute $\lambda'_w := \left\langle F_{K_1}(w), \sum_{f \in \mathbf{f}'} \mathrm{IH}(G_{K_2}(f)) \right\rangle$. Then perform a standard Merkle tree verification using λ'_w as the leaf, π as the path and st as the root. If the verification succeeds, output 1, else output 0.

Figure 6: A dynamic search authenticator scheme.

on the updates not aborting, this results in a new state st'. If some pair ρ_i and ρ'_i have equal root hashes but different children nodes, $\mathcal{B}$ outputs the children nodes as a collision for H. Otherwise, it checks if $\mathbf{f}'$ and $\mathbf{f}_w$ induce a collision on G and if so it outputs $(\mathbf{f}', \mathbf{f}_w)$. If not, it finds a collision in the authentication path π and returns that as a collision for either H or IH (depending on whether the collision occurs at a leaf or higher up the path).

By our original assumption, with non-negligible probability, $\mathcal{A}$ will output (1) $(\rho'_1, \ldots, \rho'_m)$ such that for all $i \in [m]$, $\mathsf{Update}(st, \rho'_i) \neq \perp$; and (2) $\mathbf{f}' \neq \mathbf{f}_w$ and π such that $\mathsf{Vrfy}(K, st', w, \mathbf{f}', \pi) = 1$. The first condition guarantees that either st' is "correct" (i.e., the same as what would be generated by updating st using the real receipts) or that there is a collision in one of the receipts. Therefore, if $\mathcal{B}$ does not find a collision in the receipts, st' must be correct and then second condition guarantees that either $\mathbf{f}'$ and $\mathbf{f}_w$ induce a collision on G or there is a collision in the path π. In any case, $\mathcal{B}$ will output a collision for either H, IH or G with non-negligible probability. □

In the following Theorem we show that our construction is hiding with respect to the following leakage functions:

1. $\mathcal{L}_1^{\mathsf{dsa}}(\delta, \mathbf{f}) = \#W$,
2. $\mathcal{L}_2^{\mathsf{dsa}}(\delta, \mathbf{f}, w) = \textsc{accp}(w)$,
3. $\mathcal{L}_3^{\mathsf{dsa}}(\delta, \mathbf{f}, f) = (\textsc{addp}(f), \textsc{dp}(f), \textsc{wrdp}(w_1), \ldots, \textsc{wrdp}(w_{\#f}))$,
4. $\mathcal{L}_4^{\mathsf{dsa}}(\delta, \mathbf{f}, f) = (\textsc{dp}(w), \textsc{addp}(f))$.

Theorem 6.3. *If F and G are pseudo-random then* DSA *is $(\mathcal{L}_1^{\mathsf{dsa}}, \mathcal{L}_2^{\mathsf{dsa}}, \mathcal{L}_3^{\mathsf{dsa}}, \mathcal{L}_4^{\mathsf{dsa}})$-hiding.*

Proof Sketch. We describe a polynomial-time simulator $\mathcal{S}$ such that for all probabilistic polynomial-time adversaries $\mathcal{A}$, the outputs of $\mathbf{Real}_{\mathcal{A}}^{\mathsf{dsa}}(k)$ and $\mathbf{Ideal}_{\mathcal{S}}^{\mathsf{dsa}}(k)$ are computationally indistinguishable. Consider the simulator $\mathcal{S}$ that adaptively simulates an authenticator $\widetilde{\alpha}$, a state $\widetilde{st}$, and $n \in \mathbb{N}$ simulated tokens $(\widetilde{\tau}_1, \ldots, \widetilde{\tau}_n)$ as follows:

- (Simulating α and st) given $\mathcal{L}_1^{\mathsf{dsa}}(\delta, \mathbf{f}) = \#W$, choose $\#W$ random strings $(\lambda_1, \ldots, \lambda_{\#W})$ each of length $2k$. Let `MT` be the Merkle tree with leaves $(\lambda_1, \ldots, \lambda_{\#W})$. Output $(\widetilde{\alpha}, \widetilde{st})$, where $\widetilde{\alpha} :=$ `MT` and $\widetilde{st}$ is the root hash of `MT`.
- (Simulating challenges) given $\mathcal{L}_2^{\mathsf{dsa}}(\delta, \mathbf{f}, w) = (\textsc{addp}_t(f), \textsc{dp}_t(f))$, check the addition and deletion pattern and let t^- be the last time this file was either added or deleted:
    1. if $t^- < t$, set $\widetilde{c}_t$ to be the challenge that was returned at time t^-
    2. if $t^- = t$, then choose a (new) leaf λ in `MT` at random and let $\widetilde{c}$ be its first k bits.
- (Simulating add tokens) given $\mathcal{L}_3^{\mathsf{dsa}}(\delta, \mathbf{f}, f) = (\textsc{addp}_t(f), \textsc{dp}_t(f), \textsc{wrdp}_t(w_1), \ldots, \textsc{wrdp}_t(w_{\#f}))$, check the addition and deletion patterns and let t^- be the last time this file was either added or deleted:
    1. if $t^- < t$, set $\widetilde{\tau}_t$ to be the token that was returned at time t^-
    2. if $t^- = t$, output $\widetilde{\tau}_t = (\tau_1, \ldots, \tau_{\#f}, \tau_{\#f+1})$ such that:
        (a) (for $1 \le i \le \#f$) τ_i is a random k-bit string if the word does not appear in a file that has been either added or deleted in the past (this can be verified using the word pattern) and is the same string that has been returned before otherwise.
        (b) $\tau_{\#f+1}$ is a random k-bit string
- (Simulating delete tokens) given $\mathcal{L}_4^{\mathsf{dsa}}(\delta, \mathbf{f}, f) = (\textsc{dp}_t(f), \textsc{addp}_t(f), \textsc{wrdp}_t(w_1), \ldots, \textsc{wrdp}_t(w_{\#f}))$, the simulation of delete tokens works analogously to that of add tokens.

□

6.3 A Dynamic Proof of Data Possession Scheme

In this Section we describe our construction of a dynamic PDP. The scheme is based on the privately-verifiable homomorphic linear authenticator proposed by Shacham and Waters in [SW08]. Unfortunately, the state is linear in the number of files (but not in the *size* of the file) and grows linearly with the number of deleted files. A detailed description of the scheme is given in Figure 7.

Let $F : \{0,1\}^k \times \{0,1\}^* \to \mathbb{Z}_p$ and $G : \{0,1\}^k \times \{0,1\}^* \to \mathbb{Z}_p$ be pseudo-random functions. Construct a dynamic proof of data possession scheme $\mathsf{PDP} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{Chall}, \mathsf{Prove}, \mathsf{Add}, \mathsf{Del}, \mathsf{Vrfy})$ as follows:

- $\mathsf{Gen}(1^k)$: let p be a k-bit prime. Sample a sequence $\mathbf{w}$ uniformly at random from $\mathbb{Z}_p^s$ and let K_1 be a random k-bit string. Output $K = (p, K_1, \mathbf{w})$.
- $\mathsf{Encode}(K, \mathbf{f})$: parse $\mathbf{f} = (f_1, \ldots, f_n)$ into a single string of m blocks each composed of s sectors in $\mathbb{Z}_p$. For $1 \le i \le m$ and $1 \le j \le s$, compute $t_{i,j} := F_K(i) + \sum_{j=1}^{s} w_j \cdot f_{i,j}$. Output $\mathbf{t} = (t_1, \ldots, t_m)$ and $st = (D, \mathsf{fb}(f_1), \ldots, \mathsf{fb}(f_n), \mathsf{lb}(f_n))$, where $D = \emptyset$ is a set, $\mathsf{fb}(f_i)$ is the block number of f_i's first block and $\mathsf{lb}(f_i)$ is the block number of f_i's last block.
- $\mathsf{Chall}(st)$: let B be the set of blocks not in any of the ranges stored in D. Sample a set Q of λ blocks at random from B and output $c = (K_c, Q)$.
- $\mathsf{Prove}(\mathbf{f}, \mathbf{t}, c)$: compute $\pi_1 := \sum_{i \in Q} G_{K_c}(i) \cdot t_i$. For $1 \le j \le s$, compute $\mu_j := \sum_{i \in Q} G_{K_c}(i) \cdot f_{i,j}$ and let $\pi_2 = (\mu_1, \ldots, \mu_s)$. Output $\pi = (\pi_1, \pi_2)$.
- $\mathsf{Add}(K, st, f)$: parse f into n_f blocks each composed of s_f sectors in $\mathbb{Z}_p$ and st into $(D, b_1, \ldots, b_{n+1})$. For $1 \le i \le n_f$, let $t_i := F_{K_1}(b_{n+1} + i) + \sum_{j=1}^{s} w_j \cdot f_{i,j}$. Output (st', t), where $st' = (D, b_1, \ldots, b_n, \mathsf{fb}(f), \mathsf{lb}(f))$ and $t = (t_1, \ldots, t_{n_f})$.
- $\mathsf{Del}(K, st, f)$: parse f into n_f blocks each composed of s_f sectors in $\mathbb{Z}_p$ and st into $(D, b_1, \ldots, b_{n+1})$. Add the range $(\mathsf{fb}(f), \mathsf{lb}(f))$ to D.
- $\mathsf{Vrfy}(K, st, c, \pi)$: parse π as $(\pi_1, \mu_1, \ldots, \mu_s)$. If $\pi_1 \overset{?}{=} \sum_{i=1}^{n} F_K(i, v_i) \cdot G_{K_c}(i) + \sum_{j=1}^{s} w_j \cdot \mu_j$ output 1, otherwise 0.

Figure 7: A dynamic proof of storage scheme.

Theorem 6.4. *If F is pseudo-random then* PDP *as described above is sound.*

The Theorem follows directly from Theorem 4.1 of [SW08] and Theorem 4.1 of [AKK09] and is omitted.

7 The CS2 System

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 $\mathcal{P}$. It returns some state information st which must 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. After interacting with the provider, the function returns either a sequence of files $\mathbf{f}_w \subseteq \mathbf{f}$ or $\perp$ if $\mathcal{P}$ did not send back the correct (encrypted) files.
- 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 f. 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.

7.1 Implementation

To demonstrate the feasibility of our algorithms, we implemented CS2 in C++ over the Microsoft Cryptography API: Next Generation (CNG) [Cry] and the Microsoft Bignum library (the core cryptographic library in Microsoft Windows). Our implementation uses the algorithms described in §5. The cryptographic primitives for our protocols were realized as follows.

- Encryption is the CNG implementation of 128-bit AES-CBC [FIP01].
- The hash function is the CNG implementation of SHA-256 [FIP08]. This function is also used as a (single-argument) 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 [BCK96]). 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.

Dynamic SSE. SSE was implemented using the file-based SSE scheme described in §6.1. 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 [Mer87] and incremental hash functions [BM97b]. Our incremental hash functions employ the multiplicative hash MuHash described by Bellare and Micciancio [BM97b]; 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. The output of the PRF is hashed (using SHA-256), and the hash values are treated as elements of a 2048-bit DSA group [FIP98]; 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.

7.2 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.

7.2.1 Micro-Benchmarks

SSE and DSA. 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.[6]

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

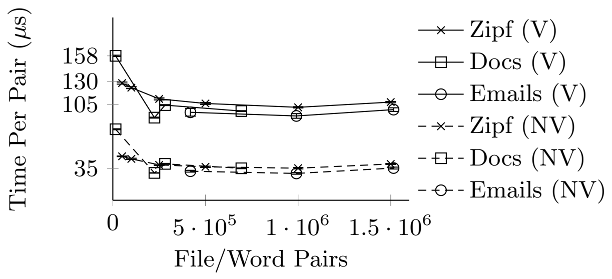

Figure 8: Execution time for generating an index for SSE.Enc, with (V) and without (NV) DSA.Auth

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 [Zip35] 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 [HD96]). 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 [Enr09]; 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 8 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 15 in §7.2.2). 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 8 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

operation	time	stddev
SSE.Search with DSA.Prove	6.3	0.2
DSA.Vrfy	54	3
SSE.AddToken	36	2
SSE.DelToken	3.0	0.2

Table 1: Execution time (in μs) per unit (word or file) for verifiable SSE operations.

verifiable SSE, we present the remainder of our micro-benchmark results only for the verifiable versions of the algorithms (however, see the full protocol results in §7.2.2 for a break down of the costs of SSE and verification).

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.[7] 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 the client-side operations of the SSE protocol and for search on the server side did 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. Table 1 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 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.[8]

Figures 9 and 10 show the costs of file addition and deletion (per unique word) and the costs of generating authenticator updates for addition and deletion (per unique word), respectively. Each of the figures shows values in a different order of magnitude, but each shows the same effects of increasing size of the authenticator tree height (note that these graphs have log-linear axes, since the size of the index is proportional to 2 to the power of the tree height). The first point in each of these figures is generated over a small number of file/word pairs, but the index and the authenticator tree are padded to size 2^{16}; this explains the higher per-word cost, since each word is attributed more of the overhead of

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

[8]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.

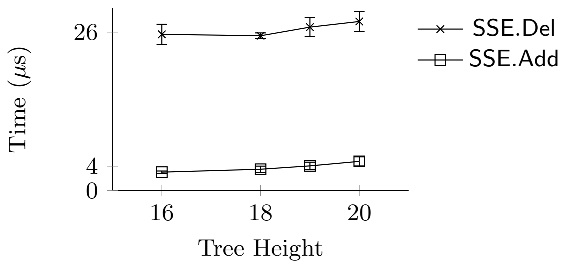

Figure 9: Per-word execution time for SSE.Del and SSE.Add.

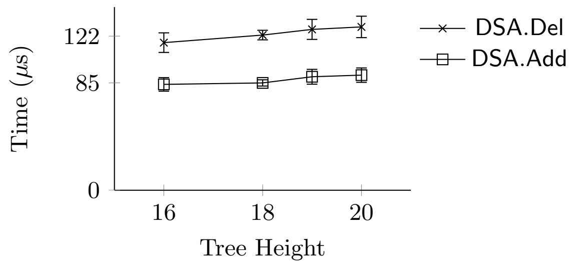

Figure 10: Per-word authenticator update computation time for DSA.Del and DSA.Add.

index construction and free list generation. The cost of writing client-generated authenticator updates to the server data structures is negligible, because it involves only memory copy operations, so we do not present results on its performance.

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.

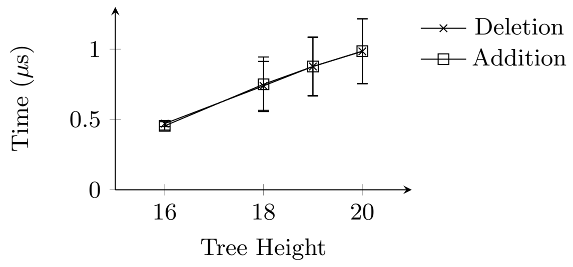

Figure 11: Per-word execution time for writing client-generated authenticator updates for file deletion and addition to verifiable data structures on the server.

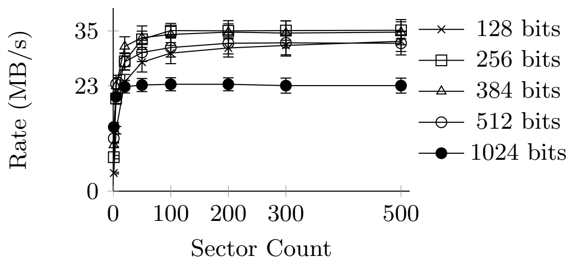

Figure 12: Comparison of PDP.Encode rates for different sector bit lengths and sector counts.

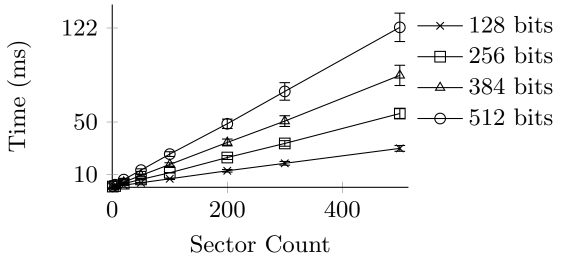

Figure 13: Comparison of PDP.Prove time for different sector bit lengths and sector counts.

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).

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 256-bit line begins to plateau. 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 the other choices.

Figure 13 shows the cost of server operations during verification. And Figure 14 shows the cost of client operations during verification for the different choices of sector size and sector count. In all cases, verification was performed over 128 randomly chosen blocks. In both figures, 256-bit sectors with 100 sectors per block take almost exactly the same amount of time as 128-bit sectors (with slightly faster performance in the 128-bit case).

Finally, to confirm that the performance of our PDP implementation was linear over different file sizes, we ran the full PDP generation algorithm for file collections ranging from 8MB to 500MB; the collections were chosen from our document data set, as before. The results were linear, so we do not provide a graph: a simple linear regression over our data shows that each additional MB of file size adds 29.4 ms to the time needed to compute the tags, as would be expected from the performance in Figure 12.

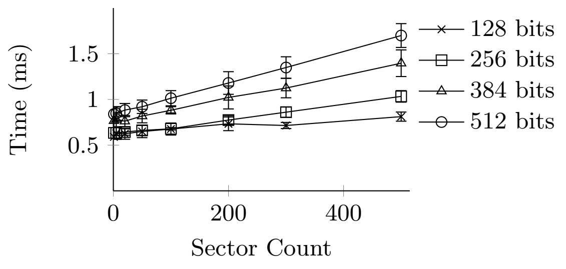

Figure 14: Comparison of PDP.Vrfy time for different bit lengths and sector counts.

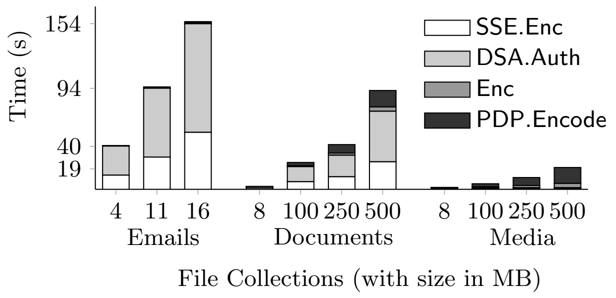

Figure 15: Execution time for STORE.

7.2.2 CS2 Performance

To evaluate the performance of CS2 as a whole, we ran the CS2 algorithms specified in Section 5 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 15 shows the results of the STORE operation, which takes the most time of any of the algorithms.[9] 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 15 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 the 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, labeled "Enc", depend only on the file size.

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

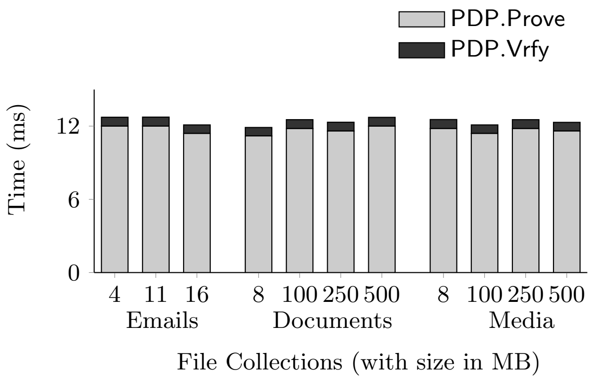

Figure 16: Execution time for CHECK.

The results of Figure 15 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.[10] The micro-benchmark results of Figure 8 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 16 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.

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.[11]

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 17 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 SSE search costs were small, even for the email index; verification time dominated the total execution time of the algorithm (and

[10]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 authenticated data structures.

[11]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.

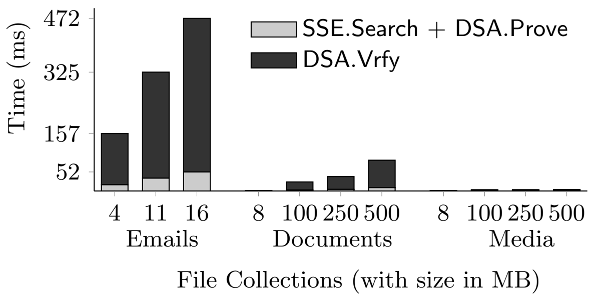

Figure 17: Execution time for SEARCH.

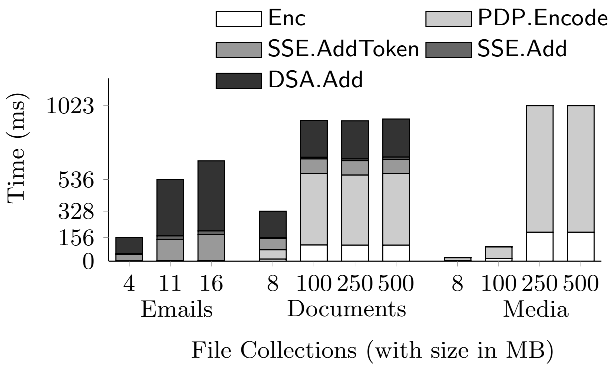

Figure 18: Execution time for ADD.

all search verification is performed on the client). However, even the longest searches with verification took less than half a second to complete and verify. And for large media collections, the search and verification time was negligibly small.

Figure 18 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, and "DSA.Add" 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, since it these costs are negligibly small. The costs of ADD fall mostly on the client: the dominant costs are PDP tag generation, SSE add token generation, 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.

A similar situation occurs in Figure 19 for the DELETE operation, which involves deleting the words from the SSE index and updating the authenticator information. The label "SSE.DelToken" refers to client generation of the delete token, "SSE.Del" refers to the server using the delete token to update the index, and "DSA.Del" 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.

For both ADD and DELETE, these results show client authenticator verification and generation to be computationally expensive. Part of this cost comes from the simple implementation of Merkle

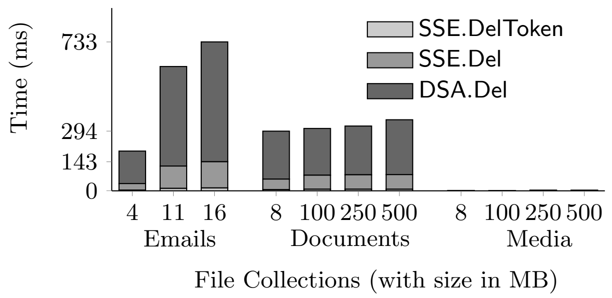

Figure 19: Execution time for DELETE.

hash trees employed by our code: in our system, the client checks or recomputes all parent hashes in the Merkle tree for each unique word being added to or deleted from a file. This cost can be mitigated in part by batched Merkle tree verification or recomputation, which performs a hash only once for each node to be checked or recomputed in the tree. But further work is needed on (more) efficient authenticator schemes.

8 Extensions

8.1 A Shared System

So far, we have described CS2 in a way that is suitable for use only by the *owner* of the data. However, we would like to be able to expand its capabilities so that it could be used by more than one client. We describe two solutions that could allow this functionality, i.e., by employing *attribute-based encryption* (ABE) or *proxy re-encryption*. Due to space constraints, we only present the version here that uses ABE. See the full version for the second solution. Both methods involve having the owner of the data encrypt each file f_i with a different key k_i (note that this is not the case in the original implementation of CS2 that uses one encryption key for all the files).

8.1.1 Using ABE

Consider the following scenario. The owner of the data, Alice, uses CS2 to securely search and update her data. However, Alice would like to allow other people that belong to a certain group to access her data (e.g., members of her family). The different symmetric keys K_i used to encrypt each seperate file f_i could be encrypted by Alice using ABE with some policy (e.g., all family members or all researchers at MSR). The output ABE encryptions are also *signed* with Alice's secret key and finally they are stored in the cloud together with Alice's data. Whenever Alice wants to give Bob the permission to search over her file collection, she sends him the appropriate token, as output by the algorithm that produces the search tokens. Bob sends the search token to the cloud, which returns the encrypted files together with the ABE encrypted symmetric keys and their signatures (computed by Alice). Assuming that Bob has a secret key with the appropriate attributes, he can decrypt the ABE encrypted key and use that to decrypt the files (note that before decryption, he also has to make sure that the signature on the ABE encrypted key verifies). By applying this method directly on CS2, we could allow Alice to share files with Bob that contain a certain keyword. However we note here that, by including the file ID as one of the keywords attached to a file, Alice can share a file by giving a search token for that file's ID.

8.1.2 Using Proxy Re-Encryption

In this case, the symmetric keys would be encrypted with a master public key PK (i.e., using the same public key for every file). Also, (as happens in the proxy re-encryption framework), all of Alice's friends would receive their own public key/secret key pair, denoted with $(\mathsf{pk}, \mathsf{sk})$. If Alice wants to allow Bob to search, she sends him the token (as output by the algorithm that produces the search tokens) and a proxy key that could be used to securely transform ciphertexts encrypted under the master public key PK to ciphertexts encrypted under Bob's public key pk. Then Bob sends the token to the cloud and receives the encrypted files together with the encryptions of the symmetric keys under the master PK. He then uses the proxy key to transform the encryptions of the symmetric keys under PK to encryptions under his own public key pk. He then decrypts these symmetric keys and uses them to decrypt the files. However, note that once Alice gives Bob the proxy key, Bob could decrypt any encryption of a symmetric key. So if he colludes with the cloud he could decrypt everything. Therefore, in order to use this approach, we should have an adversarial model that assumes that all the clients are trusted and will not collude with the cloud.

Acknowledgements

The authors are grateful to Jason Mackay for writing the indexer that was used in the experiments and to Melissa Chase and Vinod Vaikuntanathan for helpful discussions. The first author is particularly grateful to Kristin Lauter for her enthusiastic support of this project and numerous discussions on cryptographic cloud storage over the past three years.

References

[ABC$^+$05] 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.

[ABC$^+$07] 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.

[AKK09] 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.

[APMT08] 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.

[BCK96] 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.

[BCW99] D. Bindel, M. Chew, and C. Wells. Extended cryptographic file system. 1999.

[BdCOP04] 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.

[Bea92] D. Beaver. Foundations of secure interactive computing. In *Advances in Cryptology – CRYPTO '91*, pages 377–391. Springer-Verlag, 1992.

[BG92] M. Bellare and O. Goldreich. On defining proofs of knowledge. In E. Brickell, editor, *Advances in Cryptology – CRYPTO '92*, volume 740 of *Lecture Notes in Computer Science*, pages 390–420. Springer-Verlag, 1992.

[BGG94] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology - CRYPTO '94*, Lecture Notes in Computer Science, pages 216–233. Springer-Verlag, 1994.

[BJO09] 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.

[BKOS07] 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.

[Bla93] M. Blaze. A cryptographic file system for unix. In *ACM conference on Computer and communications security (CCS '93)*, pages 9–16. ACM, 1993.

[Bla94] M. Blaze. Key management in an encrypting file system. In *USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 3–3. USENIX Association, 1994.

[Blo70] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM97a] 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.

[BM97b] 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.

[BW06] 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.

[BW07] 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.

[Can00] R. Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology*, 13(1), 2000.

[CCH$^+$10] R. Canetti, S. Chari, S. Halevi, B. Pfitzmann, A. Roy, M. Steiner, and W. Venema. Composable security analysis of os services. Technical Report 2010/213, IACR ePrint Cryptography Archive, 2010. See http://eprint.iacr.org/2010/213.

[CCSP01] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for unix. In *USENIX Annual Technical Conference (FREENIX Track '01)*, pages 199–212. USENIX Association, 2001.

[CGKO06] 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.

[CGKO11] 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.

[CK10] 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.

[CKS09a] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. In *International Conference on Dependable Systems and Networks (DSN '09)*, pages 494–503, 2009.

[CKS09b] C. Cachin, I. Keidar, and A. Shraer. Fork sequential consistency is blocking. *Information Processing Letters*, 109:360–364, 2009.

[CM05] 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.

[Cor] Microsoft Corp. Winfs. http://blogs.msdn.com/b/winfs.

[Cor03] Microsoft Corp. Encrypting file system in Windows XP and Microsoft Windows Server 2003. http://technet.microsoft.com/en-us/library/cc700811.aspx, 2003.

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

[DVW09] 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.

[EKPT09] 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.

[Enr09] Enron email dataset. See http://www.cs.cmu.edu/~enron/, 2009.

[FFS88] U. Feige, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. *Journal of Cryptology*, 1(2):77–94, 1988.

[FIP98] FIPS 186-1. Digital Signature Standard (DSS). Federal Information Processing Standard (FIPS), Publication 186-1, National Institute of Standards and Technology, December 1998.

[FIP01] FIPS 197. Advanced Encryption Standard (AES). Federal Information Processing Standard (FIPS), Publication 197, National Institute of Standards and Technology, November 2001.

[FIP08] FIPS 180-3. Secure Hash Standard (SHS). Federal Information Processing Standard (FIPS), Publication 180-3, National Institute of Standards and Technology, October 2008.

[FKK06] 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.

[Fu99] K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, MIT, June 1999.

[Gia98] D. Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers Inc., 1998.

[GJSO91] 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.

[GL91] S. Goldwasser and L. Levin. Fair computation of general functions in presence of immoral majority. In *Advances in Cryptology – CRYPTO '90*, pages 77–93. Springer-Verlag, 1991.

[GM99] 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.

[GO96] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[Goh03] E-J. Goh. Secure indexes. Technical Report 2003/216, IACR ePrint Cryptography Archive, 2003. See `http://eprint.iacr.org/2003/216`.

[GPTT08] 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.

[GR10] J. Gantz and D. Reinsel. The digital universe decade - are you ready? `http://idcdocserv.com/925`, 2010.

[GRC$^+$07] 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.

[GSMB03] 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.

[GTT11] M. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60(3):505–552, 2011.

[HD96] W. Hörmann and G. Derflinger. Rejection-inversion to generate variates from monotone discrete distributions. *ACM T. Model. Comput. S.*, 6(3):169–184, 1996.

[JK07] 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.

[KK05] 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.

[KL08] J. Katz and Y. Lindell. *Introduction to Modern Cryptography.* Chapman & Hall/CRC, 2008.

[KRS$^+$03] 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.

[LKMS04] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *Symposium on Opearting Systems Design & Implementation (OSDI '04)*, pages 9–9. USENIX Association, 2004.

[LPWM09] 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.

[Mer87] 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.

[Mer89] R. Merkle. A certified digital signature. In *Advances in Cryptology - CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

[MKKW99] D. Mazières, M. Kaminsky, F. Kaashoek, and E. Witchel. Separating key management from file system security. In *ACM symposium on Operating systems principles (SOSP '99)*, pages 124–139. ACM, 1999.

[MND$^+$04] 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.

[MR92] S. Micali and P. Rogaway. Secure computation (abstract). In *Advances in Cryptology - CRYPTO '91*, pages 392–404. Springer-Verlag, 1992.

[MS02] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Principles of distributed computing (PODC '02)*, pages 108–117, 2002.

[NN98] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proc. 7th USENIX Security Symposium*, pages 217–228, Berkeley, 1998.

[NR05] 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.

[PLM$^+$10] 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.

[PR03] Y. Padioleau and O. Ridoux. A logic file system. In *USENIX Annual Technical Conference*, pages 99–112, 2003.

[PT07] 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.

[PTT08] 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.

[SCC$^+$10] 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.

[SM09] M. I. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Workshop on Hot Topics in Operating Systems (HotOS '09)*. USENIX Association, 2009.

[SW08] H. Shacham and B. Waters. Compact proofs of retrievability. In *Advances in Cryptology - ASIACRYPT '08*. Springer, 2008.

[SWP00] 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.

[Tam03] R. Tamassia. Authenticated data structures. In *Proc. European Symposium on Algorithms*, volume 2832 of *LNCS*, pages 2–5. Springer-Verlag, 2003.

[WBDS04] 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.

[WMZ03] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210. USENIX Association, June 2003.

[XKTK03] Z. Xu, M. Karlsson, C. Tang, and C. Karamanolis. Towards a semantic-aware file store. In *Hot Topics in Operating Systems (HotOS '03)*, pages 31–31. USENIX Association, 2003.

[ZBS98] E. Zadok, I. B**u**adulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998. `www.cs.columbia.edu/~library`.

[Zip35] G. K. Zipf. *Psycho-Biology of Languages.* Houghton-Mifflin, Boston, 1935.