Dynamic Searchable Encryption with Access Control

Johannes Blömer, Nils Löken

Abstract

We present a searchable encryption scheme for dynamic document collections in a multi-user scenario. Our scheme features fine-grained access control to search results, as well as access control to operations such as adding documents to the document collection, or changing individual documents. The scheme features verifiability of search results. Our scheme also satisfies the forward privacy notion crucial for the security of dynamic searchable encryption schemes.

1 Introduction

Searchable encryption [17] allows users to remotely store their data in the cloud in a secure, i.e. encrypted, fashion without losing the ability to search their data efficiently. Particularly, the data can be searched without downloading it to search it locally, and without revealing the plaintext to the cloud. Since the introduction of searchable encryption, many searchable encryption schemes with various properties have been proposed. Many schemes are single user systems, others allow multiple users, either on the data creation or on the data usage side, or both. There are schemes that allow updates to the searchable document collection and those that do not. Some schemes allow for users to verify the correctness of search results. Some of the multi-user schemes feature access control, so users can only find documents in search results that they are allowed to access. Other multi-user schemes include all relevant documents in a search result, ignoring access restrictions to documents.

Although many features for searchable encryption schemes have been proposed in the literature, schemes rarely implement more than one or two of the above features. Due to specialized constructions, it is also rather hard to combine features into a single scheme in a straightforward manner.

We construct a searchable encryption scheme that allows for many users with a multitude of different access rights to jointly maintain and update a remotely stored document collection. The collection is searched remotely, and users are guaranteed the correctness of search results. Both, the search and the update processes, respect users' access rights. The server that stores the document collection neither learns the documents it stores, nor does it learn what users search for.

Our construction assumes a multi-authority attribute-based encryption scheme that is secure against adaptive adversaries. It must be noted that, to the best of our knowledge, schemes that satisfy all our requirements have not been presented in the literature yet. However, schemes secure against static adversaries (making all their oracle queries at once) have been presented, so the assumption of an adaptively secure variant is not too far-fetched.

Related work. Research on searchable encryption started with the seminal paper of Song et al. [17]. Since then, various flavors of searchable encryption have emerged. For a comprehensive overview see the survey on searchable encryption by Bösch et al. [4].

One direction of searchable encryption research focuses on the symmetric variant (called SSE), heavily influenced by the work of Curtmola et al. [9]. SSE mainly addresses the single user setting, although multi-user extensions have been proposed. The multi-user property is achieved by replacing some applications of symmetric encryption in the original schemes by broadcast encryption [9] or attribute-based encryption [11]. The main advantage of SSE is its efficiency, which not only stems from the use of symmetric primitives, but also from the use of elaborate index data structures.

An alternative to SSE is public key encryption with keyword search (PEKS) as introduced by Boneh et al. [3]. PEKS allows multiple data sources to produce ciphertexts that only a designated user can search. Due to multiple sources, data structures for efficient search are hard to use with PEKS. Instead, typical PEKS schemes create tags for each keyword that a message contains, and append these tags to the message ciphertext. For search, each tag has to be checked whether it matches the search query. The simplicity of the tag-based approach allows for easy additions of documents to the encrypted searchable document collection. Variants of PEKS include attribute-based encryption with keyword search as presented by Sun et al. [19] and Zheng et al. [27]. Zheng et al. rely on the tag-based approach, but introduce additional data structures to make the completeness of search results verifiable; however, these data structures are static, so documents cannot be added to the document collection.

Zhang et al. [25] have identified document dynamics in most PEKS schemes, but also some SSE schemes, as a huge threat to security in practice. The threat stems from file injection attacks, allowing an adversary to create new searchable ciphertexts to which old search queries can be applied. As a result, adversaries can rather easily determine what users search for. Schemes that do not allow new ciphertexts to be found via old queries are called forward private [18]. Bost [5] puts forth a forward private dynamic SSE scheme that has verifiable search results. These results are further improved by Etemad et al. [10]. However, the schemes of Bost and Etemad et al. remain in the single user setting or do not consider access control.

In the multi-user setting, both, the SSE and PEKS-based approaches to searchable encryption allow for varying degrees of fine-grained access control to data and search results to be enforced. This can be achieved through data structures [1] or attribute-based encryption (see above). However, typically only read access to data is considered, whereas write access is ignored.

Our contribution. In this paper, we provide a searchable encryption scheme in the *multi-user* setting with *fine-grained access control*, *document collection dynamics*, and *verifiable search results*. Particularly, the set of remotely stored searchable documents can be updated and extended. Our scheme provides fine grained access to both, read and write access. With respect to document collection dynamics, our scheme provides *forward privacy*. While the individual properties of our searchable encryption scheme are not new, their *combination into a single scheme* is. Our scheme is proven secure in the random oracle model.

Paper organization. In the upcoming section, we provide a brief overview of our model and techniques for dynamic searchable encryption with access control. In Section 3, we formally define dynamic searchable encryption with access control, discuss its security,

and present building blocks for constructing it. Our searchable encryption scheme can be found in Section 4. In Section 5, we provide a brief discussion and conclude the paper.

2 Our approach to searchable encryption

In this section we describe our model for searchable encryption and provide examples. Particularly, we explain the functional properties of our searchable encryption primitive, and then continue to describe, in a general way, how we implement these properties.

2.1 Our scenario

Our goal is to construct a searchable encryption scheme in a multi-user setting. Users can search a common document collection for arbitrary (but single) keywords, or rather, the parts of the collection the respective users have (read) access to.

As an example, consider the following scenario featuring three users Alice, Bob and Charly. Alice holds access rights $\{a\}$, Bob holds access rights $\{b\}$, and Charly holds access rights $\{b, c\}$. The searchable document collection consists of two documents. Document d_1 contains keywords w_1 and w_2; document d_2 contains keyword w_1. Both documents are (read) accessible to users holding access rights $\{b\}$. Clearly, if Bob searches for keyword w_2, he should get document d_1 as his search result. Similarly, Charly can expect to get d_1 and d_2 as results to a search for w_1. However, if Alice searches for either w_1 or w_2, her search results should be empty, because Alice is disallowed from accessing either document.

Assume Alice's search result for w_1 contained d_1, but, due to access control, Alice is incapable of accessing the document. Besides not being able to access the document, Alice still would have learned something about the contents of the document. Therefore, search and access control need to be tightly coupled, and access rights must be considered during search result computation.

Our construction not only allows users to search a document collection. Users are also capable of contributing to the document collection by adding and modifying documents. We call this document collection dynamics. Similar to read access, writing documents is subject to fine-grained access control. We distinguish two types of write access: write access to the documents, and ownership, i.e. write access to the write access policy.

Regarding document collection dynamics, we prevent the removal of documents from the searchable document collection. This includes overriding documents through modification. In particular, every version of a document is kept forever. This limits the damage malicious users can cause, i.e. permanently damaging the collection via deletion of documents is prevented.

We illustrate the above points in our example scenario, extending it by separate access rights for different operations, and by version numbers to model the modification of documents over time. Furthermore, we adopt a shorthand notation for access rights that we use throughout this paper. An entry of our document collection is a 5-tuple $(d{:}v, \mathrm{r}{:}\mathbb{A}_{\mathrm{read}}, \mathrm{w}{:}\mathbb{A}_{\mathrm{write}}, \mathrm{o}{:}\mathbb{A}_{\mathrm{own}}, KW)$, meaning that the tuple represents the v-th version of document d, the document has read access policy $\mathbb{A}_{\mathrm{read}}$, write access policy $\mathbb{A}_{\mathrm{write}}$ and ownership access policy $\mathbb{A}_{\mathrm{own}}$, and it contains keywords KW. Then, our example document collection consists of tuples $(d_1{:}1, \mathrm{r}{:}\{b\}, \mathrm{w}{:}\{a\}, \mathrm{o}{:}\{b\}, \{w_1, w_2\})$ and $(d_2{:}1, \mathrm{r}{:}\{b\}, \mathrm{w}{:}\{b\}, \mathrm{o}{:}\{c\}, \{w_1\})$. Here, for illustrative purposes, the access policies are given as sets of access rights a user needs to hold in order to gain access to data.

Now, Alice updates the existing document collection by adding a new tuple $(d_1{:}2, \mathrm{r}{:}\{c\}, \mathrm{w}{:}\{a\}, \mathrm{o}{:}\{b\}, \{w_1, w_3\})$. This tuple represents a new version of document d_1. Compared to

the previous version, the new version has a different read policy and a different keyword set. Alice is capable of adding this tuple because her access rights satisfy the write policy of the tuple representing the previous version of the document. Neither Bob nor Charly are capable of performing this particular update, because their access rights do not satisfy the write policy. However, while Alice is capable of changing the document's read policy, she cannot change its write or ownership policies, because Alice's access rights do not satisfy the document's ownership policy. Note that Charly's access rights satisfy the document's ownership policy. Still, he is not capable of writing the document, but he is capable of changing the write policy.

Since we do not delete old versions of a tuple, if Charly now searches for w_1, he will find documents d_1:1, d_1:2 and d_2:1 in his search result.

Alongside access control to users' operations, we establish verifiability of these operations. This means, users are able to determine whether the server that stores the document collection and performs search on user's behalf, has tampered with the collection and has performed search properly. This necessitates users to keep a state.

However, in our setting, we do not only consider the server to be a potential threat, but also consider the impact of malicious users. Particularly, a misbehaving user may cause a (protocol abiding) server to behave in a way that verification of certain operations fails. Therefore, we introduce the concept of conflict resolution, by which the server can blame a user's malicious actions on the respective user. A trusted party, called judge, checks the server's claims of a user's wrongdoing, and issues a notification if wrongdoing is detected. We introduce notifications that enable users to distinguish whether verification of an operation failed because the server was unwilling to perform the operation properly, or some user's actions caused the verification to fail. In order for the judge to be able to identify misbehaving users, all users need to enroll before they can participate in our system.

For our security notion, we adopt the rather strong model of Löken [13], although we consider a dynamic version thereof. That means, an adversary controls the server as well as corrupt users. This model allows for attacks that no adversary can reasonably perform if it only controls either the server or corrupt user. For example, a corrupt server is unable to launch a dictionary attack against our system, e.g. in order to determine the set of keywords existing in the document collection, because the server lacks the cryptographic keys for that. On the other hand, if some user was corrupt and tried to launch a dictionary attack, the need for the user to be protocol abiding in terms of their communication behavior with an honest server limits the efficiency of the attacks. Additionally, an honest server may thwart the attack by raising alarms and limiting the number of queries the corrupt user can make; in practice such behavior would be triggered by mechanisms intended to defeat DDoS attacks. In Löken's model, corrupt servers and users are explicitly allowed to cooperate and together they are able to launch a dictionary attack: the user provides the necessary cryptographic keys, while the server is not subject to query limitations, and can operate directly on data, so it does not need to adhere to the prescribed communication behavior. The queries may also not show up in log files, so the collusion attack may go unnoticed.

For notational purposes, we group together all data stored by the server, i.e. the searchable document collection, as well as auxiliary data structures, and denote it as the server's state. For technical reasons, the server's state needs to be initialized by an enrolled user. For updates to the document collection, we adopt the notion of batch updates, i.e. users submit several new and updated documents in batches, rather than individually.

2.2 Our techniques

Our construction of dynamic searchable encryption with access control relies on the SEAC scheme of Löken [13]. SEAC provides searchable encryption with access control for static document collections. Hence, SEAC addresses the setting laid out in the first part of the above scenario. It does not admit additions of documents to the document collection or modifications of documents. Since SEAC's document collection is static, SEAC does not handle versions, write access and ownership of documents. Instead, the searchable collection is set up once and for all.

On a technical level, SEAC pre-computes all potential search results in a preprocessing phase; one search result for each combination of keyword and access policy. During search, the server outputs those search results for the searched keyword that the user, on whose behalf search is executed, has access to.

SEAC assumes an honest setup, so it does not provide means of conflict resolution. It also lacks verifiability of search results and user operations. We, thus, improve upon SEAC by achieving document collection dynamics, forward privacy, verifiability of operations, fine-grained write access, and conflict resolution. For more details on SEAC, see Section 3.2.4.

We achieve document collection dynamics by maintaining multiple SEAC collections, one for each batch-update. Since these collections are static, users, during the update procedure, sign all pre-computed search results from the update. The signatures are used in conflict resolution to blame malicious updates on the respective user, but also allow (other) users to verify the correctness of search results on a per-batch basis.

We use authenticated dictionaries to enable users to check that they have received all batch-specific results for the keyword they search for. These checks are performed relative to a digest of the data structure that is stored as part of the users' states. Additional authenticated dictionaries are used for managing document versions and fine-grained write access in a verifiable manner.

3 Formalism and building blocks

In this section, we formally define and discuss dynamic searchable encryption schemes with access control, as described in Section 2. Our discussion includes a discussion of the primitive's security properties. Additionally, we introduce the building blocks that we use in our construction of a dynamic searchable encryption scheme with access control.

3.1 Dynamic searchable encryption with access control

As described in Section 2, our notion of dynamic searchable encryption with access control features two main operations, namely an operation for searching a document collection and an operation for updating the collection. Our notion also features helper operations: a system setup operation, an enrollment operation for users, an initialization operation for the server, and conflict resolution.

Definition 1. *A* dynamic searchable encryption scheme with access control *is a 6-tuple* (Setup, UserJoin, Init, Update, Search, Resolve) *of protocols that involve five types of parties: users, a server, a key issuer, a judge and a trusted party for system setup.* Setup *is the only protocol that involves a single party.*

Setup *takes security parameter* 1^Λ *and outputs a public key PK, a secret key MSK for the key issuer, and a secret key JSK for the judge. This algorithm is executed by a* trusted party.

UserJoin *involves a* prospective user *and the* key issuer. *Both parties take in PK; the user additionally takes in her desired attribute set U; the key issuer additionally takes in MSK and the set CL of all user certificates. The key issuer outputs an updated set* CL' *of user certificates; the user outputs a state* st_{uid} *and a secret key* usk_{uid}.

Init *involves an enrolled* user *uid and the* server. *Both parties take in PK and set CL of all user certificates; the user additionally takes in* usk_{uid} *and* st_{uid}*; the server additionally takes in its state* st_{server}. *The server outputs an updated state* st'_{server}*; the user outputs an updated state* st'_{uid}.

Update *involves a* user *uid and the* server. *Both parties take in PK and CL; the user additionally takes in* usk_{uid}, st_{uid} *and a batch B of documents;*[1] *the server additionally takes in state* st_{server}. *The server may reject the update. Both parties output updated states* st'_{uid} *and* st'_{server}*, respectively.*

Search *involves a* user *uid and the* server. *Both parties take in PK and CL; the user additionally takes in* usk_{uid}, st_{uid} *and a keyword kw; the server additionally takes in* st_{server}. *The server may reject the search request. The user privately outputs a set X and updated state* st'_{uid}*; the server outputs updated state* st'_{server}.

Resolve *involves the* server *and the* judge. *Both parties take in PK and CL; the server additionally takes in* st_{server}*; the judge takes in JSK. The server outputs an updated state* st'_{server}*; the judge may output one or more notifications ntf.*

For correctness*, we require for every properly set up system, joined user, and initialized server, for every document doc that occurs in a document batch B used in any* Update *not rejected by the server, for every keyword kw: upon executing non-rejected* Search *between enrolled user uid and the server with the user using kw as her input resulting in user's output X, we have either (1) kw does not occur in doc, or (2) kw occurs in doc and doc's read policy is not satisfied by uid's attributes, or (3) doc occurs in X, or (4) doc's occurrence in X is replaced by a notification ntf that results from an execution of the* Resolve *protocol.*

Our correctness notion ensures that every document that contains a searched keyword is contained in the search result for that keyword provided that the document is accessible to the user on whose behalf search is performed, and that the update that has introduced the document to the document collection has not been found to be malicious.

Verifiability and fork consistency. The correctness notion of dynamic searchable encryption with access control is only meaningful if the server follows its prescribed protocols. *Verifiability* allows users to check that the server indeed abides by its protocols, i.e. correctness holds in the presence of dishonest users and servers. Due to the incremental nature of updates, verifiability has a prerequisite; it assumes a linear history of updates, captured by the notion of *fork consistency.* Intuitively [6], fork consistency means that if a fork occurs, i.e. two server states that are the result of applying two different updates to the same predecessor state, honest users can only ever observe updates that occur on one

[1] A set of tuples as in Section 2.1, i.e. tuples consisting of a document plaintext and version number, access policies for read and write access and ownership, and a set of keywords contained in the plaintext.

of the forks. This particularly means that two *forks cannot be merged* using reasonable amounts of resources.

In terms of our example scenario, assume Bob to update document d_2, i.e. Bob adds a tuple $(d_2{:}2, \mathrm{r}{:}\{b\}, \mathrm{w}{:}\{b\}, \mathrm{o}{:}\{c\}, \{w_2, w_3\})$. Then, fork consistency means that if the server successfully tricks Charly into thinking that Bob's update has not occurred, then Charly will never again see any update performed by Bob, because Bob's update occurs on a fork that is different from Charly's fork.

We do not formalize verifiability and fork consistency notions for dynamic searchable encryption, but do so in the context of authenticated dictionaries (see below). We make use of these dictionaries, such that their fork consistency and verifiability implies fork consistency and verifiability of our dynamic searchable encryption scheme. Our intuition about these notions is sufficient to check that our construction of dynamic searchable encryption with access control satisfies fork consistency and verifiability.

Data confidentiality. We take a more formal approach towards what malicious adversaries can learn from participating in dynamic searchable encryption with access control. The adversary we consider has full control over the server and can adaptively corrupt users. Our formulation of data confidentiality limits what such adversaries can learn.

What adversaries can learn from participating in dynamic searchable encryption with access control is captured in so called leakage functions. These functions are stateful with a shared state, i.e. an evaluation of either function depends on the inputs and outputs of all previous evaluations of all leakage functions.

Since the leakage functions describe what can be learnt from participating in dynamic searchable encryption with access control, we have one leakage function for each interaction that either involves the adversary as a participating party or is observable by the adversary. The server is always assumed to be adversarially controlled, so operations Init, Update, Search and Resolve are of the former kind. User enrollment belongs to either kind, depending on whether the user is corrupt. Enrollment of an honest user can only be observed by the adversary; it is observable via publication of the user certificate. Enrollment of a corrupt user is of the former kind, due to the adversary potentially being able to adversarially choose parts of the prospective user's key. Lastly, we also have to consider the knowledge that the adversary can gain from corrupting a once honest user. As a result, we have seven leakage functions.

We employ the leakage functions in a simulation-based security definition. An adversary gets to interact with either a real instantiation of the system with real parties using real data, or a simulator that only obtains the leakage of interactions. In either case, the adversary fully controls the server and all corrupt users. Interactions between the adversary and honest parties or the simulator, respectively, are modelled through oracles. Honest users can be enrolled via oracle $\mathcal{O}_{\mathrm{HJ}}$, corrupt users can be enrolled via $\mathcal{O}_{\mathrm{CJ}}$, and honest enrolled users can be corrupted via $\mathcal{O}_{\mathrm{UC}}$. An honest user can be made to initialize the server via oracle $\mathcal{O}_{\mathrm{Ini}}$, to perform search via $\mathcal{O}_{\mathrm{Sea}}$, and to update the document collection via $\mathcal{O}_{\mathrm{Upd}}$. Interaction with the judge for conflict resolution can be triggered via oracle $\mathcal{O}_{\mathrm{Res}}$.

In the real instantiation, the experiment plays the part of all honest users and trusted parties, and has access to all arguments passed to the oracles. In the simulation, arguments of oracle queries are passed to the respective leakage functions, and the leakage is given to the simulator. Using the leakage, the simulator attempts to imitate the input and output behavior of all honest parties communicating with each other and with the adversary, and thus, to emulate the behavior of the oracles in the real instantiation. The goal is for the

adversary to distinguish whether it interacts with the real or the simulated system. If the adversary is unable to distinguish the two setups, it does not learn more than the leakage, i.e. all data passed to the leakage functions that is not output by the leakage functions remains confidential. The two experiments are as follows.

Experiment $\mathbf{Real}_{\Pi,\mathcal{A}}(\Lambda)$:

Setup: run $(PK, MSK, JSK) \leftarrow \mathsf{Setup}(1^\Lambda)$ and give PK to adversary $\mathcal{A}$.

Queries: adversary $\mathcal{A}$ adaptively queries its oracles $\mathcal{O}_{\mathrm{HJ}}(U)$, $\mathcal{O}_{\mathrm{CJ}}()$, $\mathcal{O}_{\mathrm{UC}}(uid)$, $\mathcal{O}_{\mathrm{Ini}}(uid)$, $\mathcal{O}_{\mathrm{Upd}}(uid, B)$, $\mathcal{O}_{\mathrm{Sea}}(uid, kw)$, and $\mathcal{O}_{\mathrm{Res}}()$, where U is a set of attributes, uid is a user identifier, B is a batch of documents, and kw is a keyword. We require $\mathcal{A}$ to only use user identifiers uid that are part of results to previous calls to either oracle $\mathcal{O}_{\mathrm{HJ}}$ or oracle $\mathcal{O}_{\mathrm{CJ}}$. A user identifier uid output as part of a result to a call to oracle $\mathcal{O}_{\mathrm{CJ}}$ or used as input to oracle $\mathcal{O}_{\mathrm{UC}}$ is marked as corrupt; a corrupt uid cannot be used as input to any oracle call.[2] Oracle $\mathcal{O}_{\mathrm{Ini}}$ can be called at most once.

Responses: Upon query

- $\mathcal{O}_{\mathbf{HJ}}(U)$**,** run protocol $\mathsf{UserJoin}$, playing both the user and the key issuer, with inputs (PK, U) and (PK, MSK), respectively; remember the user's outputs usk_{uid} and st_{uid} for future use, where uid is as in the certificate (uid, uvk_{uid}) published by the key issuer.
- $\mathcal{O}_{\mathbf{CJ}}()$**,** play the key issuer's part of $\mathsf{UserJoin}$ with input (PK, MSK); $\mathcal{A}$ plays the prospective user's part of the protocol.
- $\mathcal{O}_{\mathbf{UC}}(uid)$**,** output previously generated user key usk_{uid} and user state st_{uid}.
- $\mathcal{O}_{\mathbf{Ini}}(uid)$**,** play the user's part of Init on input $(PK, CL, usk_{uid}, st_{uid})$; $\mathcal{A}$ plays the server's part; store new user state st'_{uid}.
- $\mathcal{O}_{\mathbf{Upd}}(uid, B)$**,** on input $(PK, CL, usk_{uid}, st_{uid}, B)$, play the user's part of protocol Update; $\mathcal{A}$ plays the server's part; store new user state st'_{uid}.
- $\mathcal{O}_{\mathbf{Sea}}(uid, kw)$**,** on input $(PK, CL, usk_{uid}, st_{uid}, kw)$, play the user's part of protocol Search; $\mathcal{A}$ plays the server's part; store new user state st'_{uid}.
- $\mathcal{O}_{\mathbf{Res}}()$**,** play the judge's part of protocol $\mathsf{Resolve}$ with input (PK, CL, JSK); $\mathcal{A}$ plays the server's part.

Oracle queries that take a user identifier as input are ignored, if the respective identifier has not been output by a prior oracle query, or is marked as corrupt.

Guess: $\mathcal{A}$ outputs a bit b that is output by the experiment.

Experiment $\mathbf{Sim}_{\Pi,\mathcal{A},\mathcal{S}}(\Lambda)$:

Setup: run simulator $PK \leftarrow \mathcal{S}(1^\Lambda)$ and give PK to adversary $\mathcal{A}$.

Queries: $\mathcal{A}$ gets oracle access as before, and with the same restrictions on user identifiers.

Responses: Upon query $\mathcal{O}_{op}(arg)$, give leakage $\mathcal{L}_{op}(arg)$ to $\mathcal{S}$, where $op \in \{\mathrm{HJ}, \mathrm{CJ}, \mathrm{UC}, \mathrm{Ini}, \mathrm{Upd}, \mathrm{Sea}, \mathrm{Res}\}$. Given leakage $\mathcal{L}_{op}(arg)$, the simulator $\mathcal{S}$ emulates the behavior of oracle $\mathcal{O}_{op}$ on input arg from experiment $\mathbf{Real}_{\Pi,\mathcal{A}}$. For the emulation, $\mathcal{S}$ can interact with $\mathcal{A}$.

[2] A corrupt user can perform operations on the corrupt server without involvement of an oracle.

Guess: $\mathcal{A}$ outputs a bit b that is output by the experiment.

Definition 2. *A dynamic searchable encryption scheme with access control Π provides data confidentiality relative to leakage functions $\mathcal{L}_{HJ}$, $\mathcal{L}_{CJ}$, $\mathcal{L}_{UC}$, $\mathcal{L}_{Ini}$, $\mathcal{L}_{Upd}$, $\mathcal{L}_{Sea}$, and $\mathcal{L}_{Res}$ if, fore every probabilistic polynomial time adversary $\mathcal{A}$, there is a probabilistic polynomial time simulator $\mathcal{S}$ such that*

$$|\Pr[\boldsymbol{Real}_{\Pi,\mathcal{A}}(\Lambda) = 1] - \Pr[\boldsymbol{Sim}_{\Pi,\mathcal{A},\mathcal{S}}(\Lambda) = 1]|$$

is negligible in the security parameter Λ, where the probabilities are over the random bits of $\mathcal{A}$, $\mathcal{S}$, and the experiments.

Forward privacy. The notion of forward privacy for dynamic searchable encryption schemes means that the server is unable to determine that a new document contains any given keyword, even if that keyword has been searched for before. Unfortunately, that notion does not fit our scenario, because we have batch updates, whereas the standard formulation assumes single documents, i.e. batches of size 1, to be processed during updates. Hence, we generalize the notion of forward privacy updates to consider batches of arbitrary size.

A dynamic searchable encryption scheme is forward private, if for any batch of documents that gets introduced to the searchable document collection, the server is unable to determine whether any document from the batch contains any given keyword, even if that keyword has been searched for before the update has occurred.

Forward-privacy has been shown to be crucial for the real world security of dynamic searchable encryption schemes [25].

3.2 Building blocks

We now present the main building blocks for our construction of dynamic searchable encryption with access control. The list of building blocks can directly be taken from Section 2: SEAC, digital signatures and append-only authenticated dictionaries. These building blocks are complemented by multi-authority attribute-based encryption for the purpose of access control to documents.

3.2.1 Signatures

Our discussion of building blocks starts with digital signature schemes and a natural, yet unusual security property. Alongside the typical unforgeability property of signatures, we want some signatures to hide the signed message until the signer decides to reveal the message (release an open value to the public). At the same time, the signature provides information on the signer, even before the message is revealed, so the origin of the signature, i.e. validity of the signature under a given signature verification key, can be checked. As a result, and in contrast to standard signature schemes, we have two verification algorithms: one for verifying the origin of a signature and one for verifying the signature itself.

Definition 3. *Formally, a signature scheme with delayed verifiability consists of three algorithms* (*Setup*, *Sign*, *VSig*, *VOrig*)

Setup *takes a security parameter 1^Λ and outputs a signing–signature verification key pair (sk, vk).*

Sign *takes sk and a message m and outputs a signature–open value pair* (σ, d).

VSig *takes vk, message m, open value d, and signature* σ *and outputs valid or invalid.*

VOrig *takes vk and signature* σ *and outputs valid or invalid.*

For correctness, we require for all security parameters 1^Λ *and all messages m: if* $(sk, vk) \leftarrow \mathsf{Setup}(1^\Lambda)$ *and* $(\sigma, d) \leftarrow \mathsf{Sign}(sk, m)$*, then (1)* $\mathsf{VSig}(vk, m, d, \sigma) = valid$ *and (2)* $\mathsf{VOrig}(vk, \sigma) = valid$.

Particularly, correctness implies that $(\mathsf{Setup}, \mathsf{Sign}, \mathsf{VSig})$ is a correct signature scheme when using (σ, d) as a signature.

For security of signature schemes with delayed verifiability, we adopt the standard security definition of existential unforgeability against adaptive chosen message attacks (euf-cma). That notion must individually hold relative to both verification algorithms, i.e. it should be hard for adversaries to find signatures (and corresponding open values) valid under a given verification key (relative to VSig or VOrig) unless they know the corresponding signing key. As mentioned, we additionally formalize a *hiding* notion for signature schemes with delayed verifiability. The notion captures the property that a signature does not leak any information (beyond length information) of the signed message, even to adversaries that know other message-signature pairs.[3]

Consider the following experiment $\mathbf{Exp}^{\text{hide}}_{\Pi,\mathcal{A}}(\Lambda)$ for signature scheme Π between a challenger and an adversary:

Setup: the challenger runs $(sk, vk) \leftarrow \mathsf{Setup}(1^\Lambda)$; vk is given to the adversary.

Phase I: the adversary gets access to a signing oracle. Upon query m to the oracle, the experiment computes $(\sigma, d) \leftarrow \mathsf{Sign}(sk, m)$ and gives σ to the adversary.

Challenge: the adversary outputs two messages m_0, m_1 of equal length. The challenger picks a bit b uniformly at random, computes $\sigma \leftarrow \mathsf{Sign}(sk, m_b)$, and gives σ to the adversary.

Phase II: same as Phase I.

Guess: the adversary outputs a guess b' on b.

Output: the outcome of the experiment is 1 if and only if $b = b'$.

Definition 4. *A signature scheme* $\Pi = (\mathsf{Setup}, \mathsf{Sign}, \mathsf{VSig}, \mathsf{VOrig})$ *with delayed verifiability is called* hiding, *if for all probabilistic polynomial time adversaries* $\mathcal{A}$

$$\left|\Pr\left[\boldsymbol{Exp}^{hide}_{\Pi,\mathcal{A}}(\Lambda)\right] - 1/2\right|$$

is negligible in the security parameter Λ*, where the probability is over the random choices of the adversary and the challenger.*

A signature scheme with delayed verifiability can easily be constructed by combining standard signature schemes and commitment schemes in a commit-then-sign fashion. Let $(\sigma, d) \leftarrow \mathsf{Sign}(sk, m)$. The signature $\sigma = (c, s)$ consists of a commitment c on the message m and a signature s on c (using the original signature scheme). The open value d is a decommit value for commitment c and message m. For algorithm VSig, one verifies that

[3]The leakage of length information can be prevented by following the hash-then-sign paradigm using hash functions with fixed output lengths.

signature component s is a valid signature on signature component c (using the original scheme) and verifies that c is a commitment of m using decommit value d. For algorithm VOrig, one verifies that signature component s is a valid signature on signature component c (using the original signature scheme).

It is easy to check that this construction satisfies our security requirements. Unforgability relative to VOrig is inherited directly from the underlying signature scheme. Unforgeability relative to VSig holds due to unforgeability of the underlying signature and the binding property of the commitment. The construction also satisfies the hiding due to the commitment's hiding property.

3.2.2 Append-only authenticated dictionaries

We commence our discussion of building blocks for dynamic searchable encryption with access control by reviewing append-only authenticated dictionaries. We rely on these data structures for two purposes. First, authenticated data structures allow users to check the correctness of search results, and second, authenticated data structures are used to manage documents and write access to documents.

It is important that the data structures we use for these purposes are authenticated. Authenticated data structures enable verifiability. We want the data structures to be append-only, because then the damage that malicious users can cause is limited, particularly, users cannot delete data.

An append-only authenticated dictionary is a data structure that is stored at a server, and users can add (append) data to the dictionary. A dictionary maps a label ℓ to a (multi-)set of values $\mathbf{v}$.[4] Initially all labels are mapped to the empty set. A user's append operation for a given label extends the (multi-)set that the label maps to. The server that stores an authenticated dictionary can prove whether or not the dictionary maps any given label to a non-empty set of values.

Tomescu et al. [20] propose authenticated append–only dictionaries that work as follows.

Definition 5. *An* append-only authenticated dictionary *features two types of parties, a server and a number of clients. The server provides four operations* (Setup, Append, ProveMemb, ProveAppend) *for clients to call.*

Setup *takes in security parameter* 1^Λ *and capacity* β*. The algorithm outputs public parameters pp and verification key VK.*

Append *takes in pp, dictionary* D*, digest* $\overline{D}$*, and a label–value pair* (ℓ, v)*. The operation fails if the data structure has reached its capacity, i.e. the new version number would be* β*. Otherwise, the operation outputs a new version* D' *of the dictionary and updated digest* $\overline{D'}$*.*

ProveMemb *takes in pp, dictionary* D *and label* ℓ*. The algorithm outputs the set* $\mathbf{v} = D[\ell]$ *and (non-)membership proof* $\pi_{\ell,\mathbf{v}}$*.*

ProveAppend *takes in pp and two versions* D_i, D_j *of a dictionary with version numbers* $i < j$*. The algorithm outputs a proof* $\pi_{i,j}$ *for the statement that version* D_j *was created from version* D_i *by* $j - i$ *many* Append *operations.*

[4] In the data structures literature, labels are typically called keys. We prefer the term "label" as to avoid confusion with cryptographic keys.

The client implements two operations (VrfyMemb, VrfyAppend) to verify the server's proofs.

VrfyMemb *takes in VK, digest $\overline{D_i}$ (associated with version D_i of the dictionary), label-values pair $\ell, \mathbf{v}$ and proof $\pi_{\ell,\mathbf{v}}$. The algorithm outputs a bit.*

VrfyAppend *takes in VK, two digests $\overline{D_i}, \overline{D_j}$ (associated with versions D_i, D_j of the dictionary, respectively), corresponding version numbers i, j and proof $\pi_{i,j}$. The algorithm outputs a bit.*

All operations run in polynomial time and are deterministic with the exception of Setup, which is probabilistic. We require append-only authenticated dictionaries to provide membership correctness *(mc) and* append-only correctness *(aoc), i.e. in all correctly set up systems,*

mc: for any non-empty sequence of Append operations that results in dictionary D with digest $\overline{D}$, given a (non-)membership proof $(\mathbf{v}, \pi) \leftarrow \mathsf{ProveMemb}(pp, D, \ell)$, the proof is of the correct type (membership vs non-membership) relative to the sequence of Append operations and $\mathsf{VrfyMemb}(VK, \overline{D}, (\ell, \mathbf{v}), \pi) = 1$.

aoc: for any non-empty sequence of m Append operations that results in dictionary version D_m with digest $\overline{D_m}$ and any non-empty sequence of (n-m) Append operations that, applied to D_m, results in dictionary version D_n with digest $\overline{D_n}$, given an append-only proof $\pi \leftarrow \mathsf{ProveAppend}(pp, D_m, D_n)$, $\mathsf{VrfyAppend}(VK, \overline{D_m}, \overline{D_n}, m, n, \pi) = 1$.

Intuitively, membership correctness means that the server upon call ProveMemb outputs all values associated with a given label added to the dictionary up to the time the function is called. The server's result and proof, together with the current dictionary digest, make VrfyMemb output 1. This holds whether or not $\mathbf{v}$ is empty. Append-only correctness means that if some authenticated dictionary version D_n has been created from version D_m through any number of Append operations, then the set of labels stored in D_m is a subset of the set of labels in D_n, and for all labels that both dictionaries store, the set of values associated with such a label in D_m is a subset of the set associated with that label in D_n.

Note that this notion of authenticated dictionaries *does not use secret keys.* The notion assumes that the server broadcasts all newly generated dictionary digests, e.g. $\overline{D_n}$, together with an append-only proof $\pi_{0,n}$, and that clients, upon receiving such a digest and proof check that $\mathsf{VrfyAppend}(pp, \overline{D_0}, \overline{D_n}, 0, n, \pi_{0,n}) = 1$, and more importantly, $\mathsf{VrfyAppend}(pp, \overline{D_i}, \overline{D_n}, i, n, \pi_{i,n}) = 1$, where $\overline{D_i}$ is the previous digest that the client has received and successfully verified. The initial digest $\overline{D_0}$ represents the empty set. Of course, when the server broadcasts $\overline{D_0}$, there is no append-only proof for it.

While we mostly follow the above formulation, we diverge from it in the following two ways.

1. We generalize function Append to take in a set of label–value pairs, instead of a single label–value pair.

2. We assume clients to request digests from the server, rather than the server broadcasting digests.

These changes do not affect correctness or security definitions.

For security, Tomescu et al. [20] put forth the following security notions.

Definition 6. *We call an authenticate append-only dictionary* membership secure *(ms),* append-only secure *(aos) and* fork consistent *(fc), respectively, if in a correctly set up system,*

ms: no ppt adversary can compute $(\overline{D}, \ell, \mathbf{v}, \mathbf{v}', \pi, \pi')$, $\mathbf{v} \neq \mathbf{v}'$, *with non-negligible probability, such that* $\mathsf{VrfyMemb}(VK, \overline{D}, \ell, \mathbf{v}, \pi) = 1 = \mathsf{VrfyMemb}(VK, \overline{D}, \ell, \mathbf{v}', \pi')$.

aos: no ppt adversary can compute $(\overline{D_i}, \overline{D_j}, i, j, \pi_a, \ell, \mathbf{v}, \mathbf{v}', \pi, \pi')$, $i < j$, $\mathbf{v} \not\subseteq \mathbf{v}'$, *with non-negligible probability such that* $\mathsf{VrfyAppend}(VK, \overline{D_i}, \overline{D_j}, i, j, \pi_a) = \mathsf{VrfyMemb}(VK, \overline{D_i}, \ell, \mathbf{v}, \pi) = \mathsf{VrfyMemb}(VK, \overline{D_j}, \ell, \mathbf{v}', \pi') = 1$.

fc: no ppt adversary can compute $(\overline{D_i}, \overline{D_i'}, \overline{D_j}, i, j, \pi_i, \pi_i')$, $\overline{D_i} \neq \overline{D_i'}$, *with non-negligible probability such that* $\mathsf{VrfyAppend}(VK, \overline{D_i}, \overline{D_j}, i, j, \pi_i) = 1 = \mathsf{VrfyAppend}(VK, \overline{D_i'}, \overline{D_j}, i, j, \pi_i')$.

The security notions consider neither the server nor users to be trustworthy.

Intuitively, membership security means that it is hard to find membership proofs for two distinct result sets for the same label and dictionary digest, i.e. an adversary cannot claim two different sets of values to be associated with the same label. Append-only security means that an adversary cannot produce membership proofs that certify a value to be associated with a label at some point and not be associated with that label at a later point, i.e. the adversary cannot remove values from the authenticated dictionary. Fork-consistency means that the adversary cannot compute append-only proofs that certify a digest to be a successor digest to two distinct older digests for the same version number of the data structure.

Recently, Tomescu et al. [20] have proposed the first append-only authenticated dictionary with succinct proofs, i.e. the size of proofs is independent of the number of elements stored inside the data structure. Previous proposals had non-succinct proofs for at least one operation on the data structure.

Authenticated data structures with unauthenticated data. We diverge from the syntax above by allowing unauthenticated data in authenticated append-only dictionaries, so the dictionary associates a label with some data that is covered by the dictionary's authentication mechanism, and some data that is not covered by the authentication mechanism. Hence, dictionary entries now are 3-tuples (label, authenticated data, unauthenticated data) instead of 2-tuples (label, value).

Authenticated dictionaries with unauthenticated data can easily be constructed by combining an authenticated dictionary with an unauthenticated dictionary, and storing the label together with the authenticated value in the authenticated dictionary, and the label together with the unauthenticated data in the unauthenticated dictionary. It is clear that guarantees can only be given with respect to the authenticated part, but that is acceptable in our scenario; the authenticated part inherits all its properties from the underlying authenticated dictionary. Our notion of an authenticated dictionary with unauthenticated data simplifies our view on data structures.

3.2.3 Multi-authority attribute-based encryption

Multi-authority [8] ciphertext-policy [2] attribute-based encryption [15] (MA-CP-ABE) has been used in the construction of searchable encryption schemes before [13], particularly in Löken's SEAC scheme that serves as a basis for our construction. In MA-CP-ABE, users

hold a set of attributes — access rights — in the form of secret keys. Ciphertexts are associated with access policies, and a user can decrypt a ciphertext only if her attributes satisfy the policy. However, multiple users are unable to combine their attributes in order to decrypt ciphertexts that neither colluding user can decrypt on her own. Attributes are managed by multiple authorities.

Recently, Löken [13] has proposed an extension of MA-CP-ABE called *authority key customization.* Authority key customization extends upon MA-CP-ABE by granting a user all attributes managed by a particular authority in the sense that the user is given a special key, the customized authority secret. Using that secret, the user can generate the desired attribute-specific keys herself as needed. Still, the user is unable to compute attribute-specific keys for other users.

We write "θ:u" to denote an attribute u managed by authority θ.

Definition 7. *Formally, an* MA-CP-ABE scheme with authority key customization *[13] features seven probabilistic polynomial time algorithms* Setup, AuthSetup, KeyGen, Enc, Dec, Customize, CustKeyGen).

Setup *takes security parameter* 1^Λ. *It outputs public parameters* pp.

AuthSetup *takes* pp *and an authority identifier* θ. *It outputs authority public key* pk_θ *and authority secret* msk_θ.

KeyGen *takes* pp, msk_θ, *user identifier* uid *and attribute name* u. *It outputs attribute-specific user key* $usk_{uid,\text{“}\theta:u\text{”}}$.

Enc *takes* pp, $\{pk_\theta\}_\theta$, *access policy* $\mathbb{A}$ *and a message* msg. *It outputs a ciphertext* ct.

Dec *takes* pp, $\{usk_{uid,\text{“}\theta:u\text{”}}\}_{\text{“}\theta:u\text{”}}$ *and* ct. *It outputs a message* msg.

Customize *takes* pp, msk_θ *and* uid. *It outputs a customized master secret* $sk_{uid,\theta}$

CustKeyGen *takes* pp, $sk_{uid,\theta}$ *and attribute name* u. *It outputs attribute-specific user key* $usk_{uid,\text{“}\theta:u\text{”}}$.

For correctness*, we require (1) attribute-specific keys output by* CustKeyGen *to be functionally equivalent to keys output by* KeyGen *and (2) in every properly set up system, if a user's set* $uak_{uid} = \{usk_{uid,\text{“}\theta:u\text{”}}\}_{\text{“}\theta:u\text{”}}$ *of attribute-specific keys satisfies an access policy* $\mathbb{A}$*, then for every message* msg*, we have* $\mathsf{Dec}(pp, uak_{uid}, \mathsf{Enc}(pp, \{pk_\theta\}_\theta, \mathbb{A}, msg)) = msg$.

Condition 2 of the correctness notion is the standard correctness notion for attribute-based encryption schemes. Condition 1 ensures that, for the correctness put forth in Condition 2, it does not matter whether the attribute-specific keys in uak_{uid} originate from the KeyGen algorithm or the CustKeyGen algorithm.

For security of MA-CP-ABE, we adopt the notion of security against chosen plaintext attacks in the presence of authority key customization (cpa-akc), that was outlined in [13]. Conceptually, the cpa-akc notion is a variant of the standard notion of security against chosen plaintext attacks, but, in addition to the KeyGen oracle, the adversary also gets access to a Customize oracle.

MA-CP-ABE cpa-akc-security is defined using the following indistinguishability experiment $\mathbf{Exp}^{\text{cpa-akc}}_{\Pi,\mathcal{A}}(\Lambda)$ for MA-CP-ABE scheme Π between a challenger and an adversary $\mathcal{A}$.

Global setup: the challenger runs $pp \leftarrow \mathsf{GlobalSetup}(1^\Lambda)$ and gives pp to $\mathcal{A}$.

Choosing authorities: $\mathcal{A}$ outputs a set N of authority identifiers and a set C of authority public keys. Authorities in N are considered honest, authorities in C are considered corrupt.

Honest setup: for all $\theta \in N$, the challenger runs $(pk_\theta, msk_\theta) \leftarrow \mathsf{AuthSetup}(pp, \theta)$. The public keys are given to $\mathcal{A}$. Let $PK \leftarrow C \cup \{pk_\theta\}_{\theta \in N}$.

Phase I: $\mathcal{A}$ adaptively queries oracles $\{\mathsf{KeyGen}_\theta(\cdot,\cdot)\}_{\theta \in N}$ and $\{\mathsf{Customize}_\theta(\cdot)\}_{\theta \in N}$.

- Upon query $\mathsf{KeyGen}_\theta(uid, u)$, the challenger runs $usk_{uid,\text{“}\theta:u\text{”}} \leftarrow \mathsf{KeyGen}(pp, msk_\theta, uid, u)$ and gives $usk_{uid,\text{“}\theta:u\text{”}}$ to $\mathcal{A}$.
- Upon query $\mathsf{Customize}_\theta(uid)$, the challenger computes $sk_{uid,\theta} \leftarrow \mathsf{Customize}(pp, msk_\theta, uid)$ and gives $sk_{uid,\theta}$ to $\mathcal{A}$.

Challenge: $\mathcal{A}$ outputs an access policy $\mathbb{A}^*$ and two messages msg_0, msg_1 of equal length. The challenger picks a bit $b \leftarrow_\$ \{0,1\}$, runs $ct^* \leftarrow \mathsf{Enc}(pp, msg_b, \mathbb{A}^*, PK)$ and gives ct^* to $\mathcal{A}$.

Phase II: is the same as Phase I.

Guess: $\mathcal{A}$ outputs a bit b'. The experiment outputs 1 if and only if (1) $b = b'$, (2) the access policy $\mathbb{A}^*$ is not satisfied by attributes controlled by corrupt authorities, and (3) for no user identifier uid, the set of attributes queried for uid together with the set of attributes controlled by corrupt authorities and the set of attributes for authorities for which a customized key for uid was queried satisfies $\mathbb{A}^*$. Otherwise, the experiment outputs 0.

Definition 8. *An MA-CP-ABE scheme Π with authority key customization is called* cpa-akc-secure *if, for every probabilistic polynomial time adversary $\mathcal{A}$,*

$$|\Pr[\boldsymbol{Exp}^{cpa}_{\Pi,\mathcal{A}}(\Lambda) = 1] - 1/2|$$

is negligible in the security parameter Λ, where the probabilities are over the random bits of the adversary and the experiment.

We are presently unaware of any scheme that satisfies the above cpa-akc notion for MA-CP-ABE with authority key customization while simultaneously featuring fully independent authorities, a property implicit in our MA-CP-ABE definition. Typical MA-CP-ABE schemes in the literature often require coordination between authorities or feature other inconvenient restrictions, i.e. authorities are not fully independent, e.g. Chase's scheme [8], or the universe of attributes must be fixed at setup time (small universe), e.g. Lewko et al.'s scheme [12]. MA-CP-ABE schemes (even without authority key customization) that have fully independent authorities and support a dynamic attribute universe (large universe) presently only satisfy a rather weak notion of security in which the adversary has to make all its KeyGen queries at once, i.e. when sending sets N and C in the above game, the adversary also sends its oracle queries from Phases I and II, and the challenge messages msg_0, msg_1; the adversary then obtains all the respective responses from the challenger; finally, the adversary outputs its guess b without making any additional query to the challenger. This weak notion of security for MA-CP-ABE schemes was introduced by Rouselakis and Waters [14], whose scheme satisfies the security notion. Löken has also shown that the scheme by Rouselakis and Waters can be extended to provide authority key customization [13].

3.2.4 SEAC

As mentioned in Section 2, our construction relies on Löken's SEAC scheme [13], which we extend, using the other building blocks mentioned above, in order to achieve dynamic searchable encryption with access control. We now briefly review the SEAC scheme. Note that SEAC only achieves searchable encryption with access control on static document collections.

Definition 9. *SEAC provides six probabilistic polynomial time algorithms* (*Setup*, *KeyGen*, *BuildIndex*, *Trpdr*, *Search*, *Dec*).

Setup *takes security parameter* 1^Λ. *It outputs public parameters pp, master secret msk and owner key ok.*

KeyGen *takes pp, msk, user identifier uid and attribute set* U_{uid}. *It outputs user key* usk_{uid}.

BuildIndex *takes pp, ok, and document collection B. It outputs index Index and ciphertext set CT.*

Trpdr *takes pp,* usk_{uid} *and keyword kw. It outputs a search query ("trapdoor")* $t_{uid,kw}$.

Search *takes pp, Index, CT and* $t_{uid,kw}$. *It outputs set* $X \subseteq CT$.

Dec *takes pp,* usk_{uid} *and ciphertext ct. It outputs a document doc.*

SEAC guarantees in every properly set up system, that for $X \leftarrow \mathsf{Search}(pp, Index, t_{uid,kw})$ *and for each* $doc \in B$, *we have (1)* $kw \notin doc$, *or (2)* U_{uid} *does not satisfy doc's access policy, or (3) there is* $ct \in X$ *such that* $\mathsf{Dec}(pp, usk_{uid}, ct) = doc$.

Correctness of SEAC is conceptually similar to, but simpler than, correctness of dynamic searchable encryption with access control: the result to a search request for keyword kw can be decrypted to the subset of documents from B that contains kw and is accessible to the user.

Security-wise, SEAC provides semantic security against malicious servers that collude with corrupt users. However, due to substantial modifications that we make to SEAC in the upcoming section, security properties of SEAC do not translate to security properties of our construction. Hence, we see no need to further discuss SEAC's security.

SEAC's index structure. SEAC employs access control such that only a user who can access a certain document may find that document in a search result presented to her. SEAC's core idea is to use a dictionary that enables fast access to pre-computed search results while also ensuring access control; this is similar to the data structures (albeit with access control) from the work of Curtmola et al. [9]. In SEAC, there is one pre-computed search result for each combination of keyword kw and access policy $\mathbb{A}$ that occurs in the document collection. Access control to pre-computed results is established by MA-CP-ABE.

As stated, SEAC uses a dictionary to provide quick access to pre-computed search results relevant for a searched keyword. The dictionary is the first component of SEAC's index structure that allows for efficient search. Figure 1 shows an example index structure computed from our example document collection from Section 2.1.

The dictionary HT holds a single label for each keyword present in the (plaintext) document collection. The associated values are the sets of pre-computed search results for the respective keyword. These sets are implemented as encrypted linked lists.

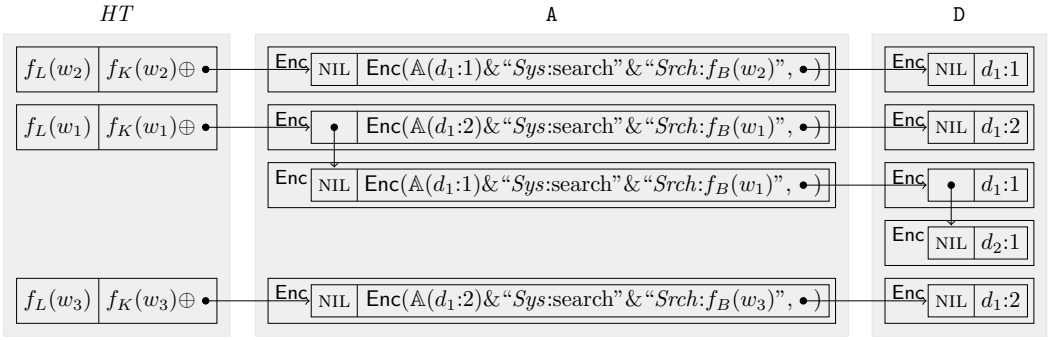

Figure 1: An example of the data structures used in SEAC, using our (post-update) example document collection from Section 2.1 (p. 3), ignoring for now that SEAC does not allow for updates. The left-most blocks represent HT entries, the blocks in the center represent `A` entries, and the right-most blocks represent `D` entries. We denote the (read) access policy of a document d as $\mathbb{A}(d)$; note that documents present in the same `D` list have the same read access policy. The figure does not show dummy entries, as to not clutter the presentation.

In order to prevent the server from gaining information about a keyword from a label, the label is derived from the keyword by applying a pseudorandom function f_L. The pointers to sets of pre-computed search results stored in HT are (symmetrically) encrypted under keys that depend on the relevant keyword; the keys are derived from keywords using pseudorandom function f_K.

Following a pointer from an HT entry, a cell in some array `A` is reached. The cell is an encrypted node from an encrypted linked list, that can be decrypted using a key that is part of the pointer to the list node. The nodes in `A` provide access control to the actual pre-computed search results, i.e. sets of (pointers to) documents that share an access policy and contain the relevant keyword. These sets of document pointers are stored in an array `D` in encrypted linked lists. As a means of access control to `D` lists, pointers to `D` lists are encrypted using MA-CP-ABE; the policies used depend on the access policies of documents pointed to in the respective `D` lists; the policies also feature a binder term derived from the relevant keyword using a pseudorandom function f_B.

The actual document ciphertexts that `D` lists nodes point to are stored in a separate ciphertext set CT not shown in Figure 1. Keys to pseudorandom functions f_L, f_K, f_B are part of users' keys as well as the key of the document collection's owner.

SEAC's inner workings. The trapdoor/search query generated by a user contains a keyword-specific label for the dictionary. In order for the server to decrypt, i.e. access, precomputed search results, the trapdoor also contains some cryptographic keys in addition to the keyword-specific label. Concerning the cryptographic keys, SEAC ensures that the server cannot use attribute-specific keys from a query to decrypt pre-computed search results for keywords other than the searched one; this is what the binder term (from function f_B) is used for. Users also retain some attribute-specific keys that would otherwise allow the server to decrypt document ciphertexts.

As a result, SEAC features three types of attributes: general attributes that are used for all access control purposes (e.g. attributes describing roles that a user may have), special attributes for accessing pre-computed search results that depend on the searched keywords, and attributes that prevent the server from using attribute-specific keys from

search queries to decrypt document ciphertexts. SEAC uses three attribute authorities $Usr, Srch, Sys$ that correspond to the three types of attributes. Authority Usr manages the general purpose attributes of users. Authority $Srch$ manages attributes that depend on the searched keyword and prevent the server from using keys from a query for some keyword to decrypt pre-computed search result for another keyword. Hence, $Srch$ manages keyword-dependent attributes. Authority key customization for MA-CP-ABE is used so that users can generate keyword-specific attributes themselves. Finally, authority Sys manages two attributes, "Sys:search" and "Sys:read", which are used during search and document decryption, respectively; users retain attribute "Sys:read" to prevent the server from decrypting document ciphertexts.

In order to realize the above functionality, when encrypting data, whether it may be documents or pre-computed search results, SEAC takes into account attributes managed by multiple authorities, while the (plaintext) document collection only considers general purpose attributes in its access policies. So, when encrypting data, SEAC extends access policies from the document collection by some additional attributes, depending on the type of data. We denote the extension of an access policy $\mathbb{A}$ by attribute u as $\mathbb{A}\&u$; the extension results in a new access policy that is satisfied by all attribute sets that satisfy policy $\mathbb{A}$ and contain the attribute u. The functionalities described above require a document d from the plaintext document collection with access policy $\mathbb{A}_d$ to be encrypted under policy $\mathbb{A}_d\&$"Sys:read"; a pre-computed search result that contains d for keyword kw from d need to be encrypted under policy $\mathbb{A}_d\&$"Sys:search"$\&$"$Srch$:$f_B(kw)$".

4 Construction

In this section, we present our dSEAC scheme for dynamic searchable encryption with access control. dSEAC implements the idea laid out in Section 2. That is, the dynamic aspect is achieved by computing static searchable SEAC collections, one collection per update. In dSEAC, for the purpose of verifiability, users need to share a consistent view on the number of updates that have been performed. We use this number of updates as a time stamp throughout the system. Users maintain a consistent view on the number of updates, and thus, have synchronized discrete clocks, due to dSEAC's reliance on authenticated data structures.

4.1 Extending SEAC

We extend the basic SEAC construction from the previous section in four dimensions, namely verifiability, time awareness and dependence, document and version management, and conflict resolution. We present these extensions in the upcoming subsections. Each extension provides progress towards one of our goals with dSEAC.

The verifiability extension enables (static) verifiability of per-SEAC-instance search results. Time awareness and dependence considers creation times of SEAC instances, which is necessary for achieving forward privacy. Document and version management is necessary because it enables documents to be updated over time. Simultaneously, it enables fine grained access control to data, particularly write access to data. Conflict resolution allows dSEAC to recover from malicious user's actions, and thus, enables dSEAC to be used in the multi-user setting, especially allowing for multiple (and even malicious) writers.

For our extensions to work, we need to introduce a couple of cryptographic keys in addition to the keys used in SEAC. As a result, a user's secret key in dSEAC consists of

keys for pseudorandom functions, attribute-specific keys and customized authority secrets (from SEAC), signing keys (for verification purposes). Users' signature verification keys are made public.

4.1.1 Verifiability

For verifiability of SEAC search results, we rely on signatures with delayed verifiability and authenticated dictionaries. We retain the search structures from SEAC (c.f. Figure 1), and enhance them by list authenticators, as shown in Figure 2.

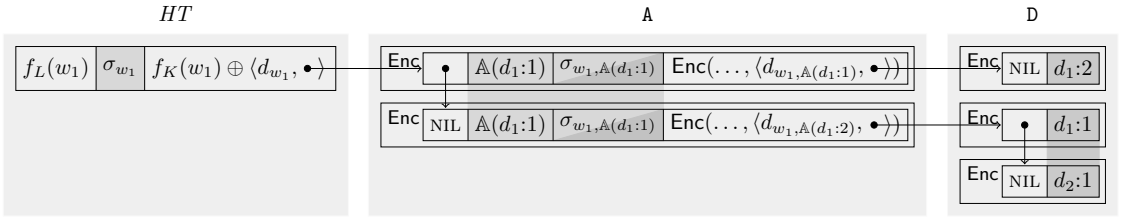

Figure 2: Changes in SEAC data structures due to verifiability. The entries for keyword w_1 have been adapted from Figure 1. Signature σ_{w_1} (marked light gray) is a list authenticator on w_1's A list. The signature authenticates all data from the A list marked light gray, i.e. the access policies of D lists, and the D list authenticators. Signatures $\sigma_{w_1,\mathbb{A}.}$ (marked dark gray) are list authenticators on the data from D lists marked dark gray, e.g. $\{d_1{:}1, d_2{:}1\}$ for the D list with access policy $\mathbb{A}(d_1{:}1)$. Note that documents from the same D list have the same (read) access policy. We store the open values that correspond to list authenticators in an encrypted form alongside the pointers to the respective authenticated lists.

List authenticators are (hiding) signatures on the (useful) data stored in the lists, but ignore administrative data required to maintain the list structure. Therefore, a list authenticators of a D list is a signature on the (ordered) set of document pointers from the list. The list authenticator of a D list is stored in an A list node alongside a pointer to the D list. The list authenticator is not MA-CP-ABE-encrypted, but its corresponding open value is. Alongside a D list authenticator, A list nodes store the access policy used to encrypt the pointer to the D list and the D list authenticator's open value. The signature that results from signing the (ordered) set of access policies and D list authenticators from an A list is used as that A list's authenticator. A list authenticators are stored alongside a pointer to the respective A lists in HT; the open value that corresponds to an A list authenticator is encrypted together with the pointer the respective A list. In contrast to before, the dictionary HT now admits authenticated and unauthenticated data, c.f. Section 3.2.2 (p. 13): (encrypted) list pointers and open values are stored unauthenticated, while the corresponding list authenticators are stored in an authenticated fashion.

Authenticating the contents of A lists and, indirectly, D lists, results in verifiability of search results. When performing search, the server simply includes a membership proof from the authenticated dictionary, list authenticators, open values, and access policies encountered during search, and document pointers in the search result. The membership proof prevents the server from tampering with any directly or indirectly authenticated data, even if the server colludes with malicious users. Users, when obtaining an search result, can use the membership proof to verify that the server has provided the correct A list authenticator. The A list authenticator is used to verify the completeness of provided D list authenticators and access policies. For that, the user applies verification algorithm VSig from the signature scheme with delayed verifiability on the list authenticator, the

corresponding open value, and the sets of access policies and (D) list authenticators. Using the provided access policies, the user checks what policies are satisfied by her attribute set: for policies satisfied by her attribute set, a D list, in the form of a set of document pointers and an open value for the list's authenticator, should be present in the search result. The D list authenticators are used to verify the completeness of provided document pointers via algorithm VSig. For access policies not satisfied by the user's attribute set, the user uses algorithm VOrig to verify the origins of list authenticators that correspond to reported non-satisfied access policies. All list authenticators are verified relative to the same signature verification key, i.e. all signatures have the same origin.

4.1.2 Time awareness and dependence

For time dependence, we apply two techniques; one is applied to the dictionary HT, the other is applied to A lists. See Figure 3 for changes to the SEAC search structure due to time dependence.

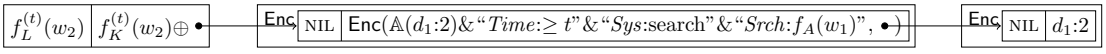

Figure 3: Changes in SEAC data structures due to creation time dependence. The entry for keyword w_2 has been adapted from Figure 1. Note that the label and the decryption key of the HT entry have been modified, so they depend not only on the keyword, but also on the creation time t. The access policy from the A list node has been modified to depend on the creation time as well. D list nodes remain unchanged.

In order to achieve forward privacy when combining multiple SEAC instances, labels and decryption keys for the same keyword must be different in different SEAC instances; otherwise, it is easy to determine that two instances share a keyword. That is why we need to make labels and keys for HT entries creation time dependent.

We achieve time dependence of HT entries by applying a technique due to Bost [5]. Essentially, without going into details, Bost's technique derives the labels and keys for HT entries from a keyword and the current time in a way that allows the server to recompute labels and keys for previous time steps, while preventing the server from predicting labels and keys for the same keyword and future time steps. Technically, this is achieved by applying a trapdoor function to a keyword-specific seed multiple times, the number of applications depends on the number of occurrences of the keyword in the document collection. However, we need to introduce a small twist to Bost's technique due to differences in settings; while we are in the multi-user setting and allow users to be malicious, Bost only considers a single user. In our multi-user setting, users may be malicious, therefore we do not want to rely on user's claims on occurrence counts of keywords in the document collection. Instead, we simply assume every keyword to occur in every time step, so in dSEAC the number of applications of the trapdoor function is the number of time steps.

In addition to time dependence of labels and keys in HT entries, we need access policies in A list nodes to be time dependent. Otherwise, the server can combine the time dependent bits from a new search query (say, at time t') with the time independent bits from an old search query (say, at time $t < t'$) for the same keyword. Then the server can use the time dependent bits to access an A list from a SEAC instance created at time t^* with $t < t^* \leq t'$, but can use the time independent bits, i.e. access rights, to access D lists created at time t^*. Particularly, the server is allowed to access D lists created at time t^* with access rights from the time t' query, but not with access rights from the time t query.

So, having access policies in `A` list nodes unaware of creation times enables the server to access data that it is not allowed to access.

In order to implement time dependent access policies, we introduce a MA-CP-ABE authority *Time*. Every user receives a customized master secret of this authority. When computing a query at time t, a user includes attributes managed by *Time* representing t in their query. Conversely, when creating a searchable document collection at time t', a user includes t' in the extended access structure used to encrypt the `D` list pointer inside `A` list nodes. This inclusion happens such that only search queries that include attributes for some time $\geq t'$ satisfy the policy. In Appendix A, we give an overview of how this comparison can be achieved efficiently.

4.1.3 Document management and versioning

SEAC uses a simple data structure for storing document ciphertexts. We replace that simple data structure with a more elaborate data structure that allows for fine-grained control over write access and ownership, while also keeping track of updates to documents. It must be noted that, while applicable to individual SEAC instances, the changes presented here only really make sense when multiple SEAC instance are combined.

Using our notation from Section 2.1, a (plaintext) document is a 5-tuple (d:v, r:$\mathbb{A}_{\text{read}}$, w:$\mathbb{A}_{\text{write}}$, o:$\mathbb{A}_{\text{own}}$, KW) consisting of (the v-th version of) a document d, read policy $\mathbb{A}_{\text{read}}$, write policy $\mathbb{A}_{\text{write}}$, ownership policy $\mathbb{A}_{\text{own}}$ and a set KW of keywords contained in the document. For our data structure, we split the information provided in these tuples into three parts: (1) information for read access, (2) information for write access and (3) information on keywords. Information on keywords are not relevant to this part of our discussion. For the other two types of information, we introduce new tuples that store the relevant information in a way that allows for fine-grained access control to be enforced.

We introduce *data tuples* for storing information for read access. That includes the document itself. Information for write access and ownership is stored in a separate tuple that we call *management tuple.* Data tuples are stored in an authenticated dictionary *Dat*, while management tuples are stored in an authenticated dictionary *Man*. Management and data tuples enforce access policies through combinations of MA-CP-ABE and signatures. Every document version is represented by one data and one management tuple. However, either type of tuple can be shared among multiple document versions, i.e. changing a document's write policy results in a new version of the document's management tuple, but leaves the data tuple unchanged. Similarly, changing a document's contents or read policy only results in a new version of the data tuple, but does not affect the management tuple.

Management and data tuples for the same document (version) have three components in common: a tuple identifier *tid*, a management tuple version number v_{mt}, and a data tuple version number v_{dt}. *tid* is also shared among all versions of the same document. v_{mt} and v_{dt} allow for cross references between tuples. Management and data tuples also each contain two (standard, i.e. non-hiding) signatures: one signature created under the signing key of the user that created the tuple, and one signature under a signing key that allows everyone to verify that the creator of the respective tuple holds certain (write or ownership) access rights.

$$(tid, v_{mt}, v_{dt}, \mathsf{ABE.Enc}(\mathbb{A}_{\text{read}}, d{:}v), uvk_{uid}, \sigma_{\text{write}}, \sigma_{uid})$$

Figure 4: The structure of a data tuple.

Figure 4 shows a data tuple. As stated above, a data tuple contains tuple identifier tid, the version number v_{mt} of the management tuple that corresponds to this data tuple, and the data tuple's own version number v_{dt}. The document itself is encrypted under the document's read policy $\mathbb{A}_{\text{read}}$ using MA-CP-ABE.[5] The data tuple contains the signature verification key uvk_{uid} of the user that created the data tuple. The tuple also contains two (standard) signatures σ_{write} and σ_{uid}. These are signatures on the data tuple under a key dsk_{write} and the user's signing key that corresponds to uvk_{uid}, respectively. The key dsk_{write} is encrypted and contained in the management tuple that corresponds to the data tuple.

$$(tid, v_{mt}, v_{dt}, \mathsf{ABE.Enc}(\mathbb{A}_{\text{own}}, dsk_{\text{own}}), \mathsf{ABE.Enc}(\mathbb{A}_{\text{write}}, dsk_{\text{write}}), dvk_{\text{own}}, dvk_{\text{write}}, uvk_{uid}, \sigma_{\text{own}}, \sigma_{uid})$$

Figure 5: The structure of a management tuple.

Figure 5 shows a management tuple. As stated before, management tuple contains a tuple identifier tid, its own version number v_{mt}, and the version number v_{dt} of the oldest data tuple to which this management tuple corresponds. The management tuple also contains document owner's (dsk_{own}) and writer's (dsk_{write}) signing keys that are encrypted under the document's ownership and write policy ($\mathbb{A}_{\text{own}}, \mathbb{A}_{\text{write}}$), respectively, using MA-CP-ABE. The corresponding signature verification keys are dvk_{own} and dvk_{write}. As with data tuples, management tuples contain the signature verification key of the user that computed the management tuple, and two signatures $\sigma_{\text{own}}, \sigma_{uid}$ on the management tuple under dsk_{own} and the signing key that corresponds to uvk_{uid}, respectively.

In both types of tuples, the σ_{uid} components ensure that the user responsible for creating the tuple can be identified, e.g. in case of wrongdoing, at least if the server abides by its protocols. MA-CP-ABE is employed to make sure that users have certain access rights (attributes) that are needed to get access to a document plaintext or a signing key. The signing keys, in combination with the access rights necessary to access these keys, certify to others that the user who created a tuple (of version number ≥ 2) actually has the necessary rights to do so. When checking a user's rights to create a new version of an existing tuple, the latest existing management tuple with the same identifier tid has to be consulted, and the new tuple's signatures have to be verified relative to the old management tuple's writer's or owner's signature verification key, depending on the tuple that is to be verified.

When storing the tuples in authenticated dictionaries, we use the tid fields of the tuples as labels for the dictionary. As with HT entries, the signatures are authenticated, while the remainder of the management and data tuples do not need to be authenticated by the authenticated dictionary itself. Their authentication is indirect due to their signatures being authenticated.

Our use of combinations of encryption and signatures for achieving fine-grained access control is inspired by the Sharoes system [16]. Attribute-based signatures may be used as an alternative to our approach for enforcing fine-grained write access, see [26] for an example; however, we opt to enforce fine-grained write access as described above, in order to keep our construction simpler, i.e. not requiring even more primitives.

[5] In practice, we would use hybrid encryption for storing document ciphertexts.

4.1.4 Conflict resolution

Conflict resolution counters malicious users creating a SEAC instance that does not adhere to the prescribed structure. The deviation may result in the server, during search, to get stuck or deliver wrong search results. For example, an `A` list node's pointer to a `D` list may be NIL, or a list authenticator may be computed on wrong data. In such cases, an honest server needs a mechanism to blame the malicious user's action on the user. Conflict resolution provides that mechanism.

Whenever the server sees a need for conflict resolution, the server submits the potentially flawed SEAC instance in question to a judge for the judge to review the instance. In addition, the server sends a description of how the flaw was discovered. Typically, this is a search query.

The judge then outputs a temporary notification that the instance is under investigation. Using the notification, the server can then proceed to serve search requests, while the judge investigates the SEAC instance, replacing search results from the instance under investigation by the judge's notification. Similarly, once the judge has concluded an investigation and found that the SEAC instance was indeed flawed, the judge issues a permanent notification that, again, replaces search results from the flawed SEAC instance.

$$(type, t_{\text{issue}}, t_{\text{affected}}, \sigma)$$

Figure 6: The structure of a notification.

In order for notifications to serve their purpose, they must state their validity period, state their purpose, i.e. what SEAC instance is under investigation or has been found to be flawed, and be verifiable. Hence, notifications are as shown in Figure 6, i.e. consist of a type (temporary or permanent), the issuance time, the affected time (SEAC instance affected), and a signature on the tuple under the judge's signing key.

Since notifications come in two flavors, temporary and permanent, we need a mechanism for temporary notifications to be invalidated. We define temporary notifications to be valid only for the time step in which the notification has been issued. While a temporary notification is valid, the server must reject all updates, presenting the temporary notification to users who attempt to perform the update. This is because, in our construction, an update advances time, and thus, invalidates the temporary notification.

4.2 Combining the extensions: dSEAC

We now present dSEAC, our dynamic searchable encryption scheme with access control. The scheme implements the ideas laid out in Section 2, and combines the changes and extensions to SEAC proposed in Section 4.1.

Specifically, we show how to integrate the previously proposed modifications to SEAC into dSEAC, present the additional data structures needed, and how the parties in dSEAC operate on the data structures. Note that the above proposals are somewhat imprecise as to not obscure the underlying mechanisms by complicated details. This especially holds for the details of changes to the internals of the search structure, as presented in Sections 4.1.1 and 4.1.2. We give these details in Appendix B, but our construction can be understood without knowledge of these details.

Syntactic changes to SEAC. The changes to SEAC make it necessary to alter the signatures of functions provided by SEAC. Due to time dependence, SEAC's BuildIndex

function needs to take the time stamp of the SEAC instance that is to be constructed into account. The time stamp is therefore taken as an additional parameter. We change the output of the BuildIndex function to only output the index itself. In our construction, we make the computation of management and data tuples explicit, rather than computing these tuples implicitly as part of BuildIndex as SEAC does. The current time stamps also need to be considered when formulating search queries, so we adapt the signature of SEAC's Trpdr function accordingly.

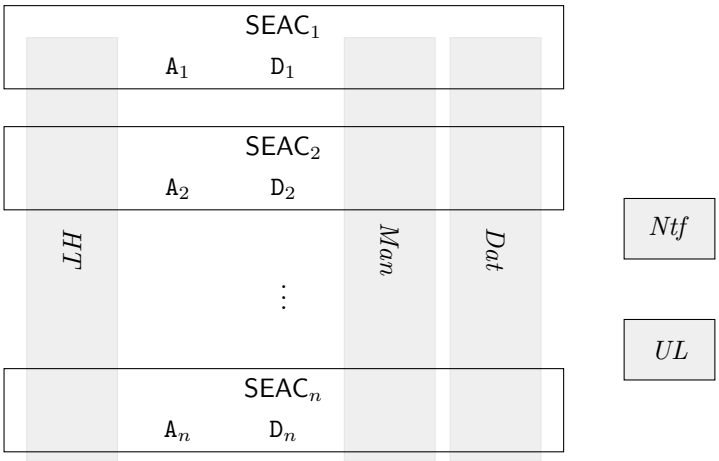

Figure 7: Data structures when combining multiple SEAC instances into a dSEAC instance by applying modifications discussed in Section 4.1.

Integration of SEAC instances. We now turn our attention towards the integration of multiple SEAC instances into dSEAC. Figure 7 displays the integration of instances on the data structure level. Every SEAC instance retains its separate A and D data structures. However, a single dictionary HT is shared among SEAC instances. Similarly, management and data tuples originating from the various SEAC instances are stored in shared dictionaries Man and Dat. While the dictionaries could be static for single SEAC instances before, they now need to allow for additions of data, because new data is added whenever a new SEAC instance is created. Since data is only ever added to these dictionaries, we implement them as authenticated append-only dictionaries.

Additionally, dSEAC uses two more data structures, an update log UL and a notification set Ntf. The update log is a set that contains, for each update that has ever been performed, a summary of the post-update system state, called "last completed operation" (variable *lco* in our scheme). The number of elements in the update log serves as a time stamp for our system, i.e. UL is a discrete clock. The notification set simply stores all notifications issued by the judge. We do not have any particular requirements on the data structures used to implement UL and Ntf.

$$(t, dig, ch, vk, \sigma)$$

Figure 8: The tuple representing the last completed operation.

The aforementioned last completed operation is stored at the server and replicated at users. The concept of the last completed operation has been adopted from Cachin and

Geisler [6]. Specifically, the last complete operation, *lco*, is a tuple as shown in Figure 8. As mentioned, *lco* is to represent a post-update system state. For that purpose, *lco* consists of a time stamp t, digests *dig* of the authenticated append-only dictionaries, a summary *ch* of the changes made by the update, a signature verification key *vk*, and a (standard) signature σ. The signature verification key belongs to the party that performed the update, and the signature σ is a signature on the other components of the tuple. The summary *ch* depends on the update. Server initialization leaves *ch* empty, in conflict resolution, *ch* is the judge's signature from the permanent notification that resulted from conflict resolution, and for other updates, *ch* is a signature on the SEAC instance computed as part of the update.

The server stores *lco* as part of its state st_{server}. Users' states contain the *lco* tuple of the latest update they have performed or verified. Particularly, when performing an update or search, users first contact the server to request the server's *lco*, verify the signature on the server's *lco*, and then ask the server to provide append–only proofs for the digests contained in the users' and the server's *lco* tuples. If the server's append–only proofs can be verified successfully, users adopt the server's *lco* as their own. When performing an update, a user computes her new *lco* from the information obtained during the update operation.

Protocols of dSEAC. We now present the protocols of dSEAC. As said before, we do not go into too many details here. The details can be found in Appendix B.

Our use of SEAC in the following description always refers to the version of SEAC that results from incorporating the modifications and extensions outlined above! While MA-CP-ABE is part of SEAC, it remains implicit. We denote signature schemes by Σ, and assume standard signatures and hiding signatures with delayed verifiability to use the same signing and verification keys (due to the relations between these primitives); we note that hiding signatures are only used as part of our augmented SEAC scheme, so in the following presentation, we do not use hiding signatures explicitly. We denote authenticated append-only dictionary schemes with unauthenticated data by AAD.

Since our presentation is quite heavy on notation, we provide a list of variables used in our construction of dSEAC. The list features brief descriptions of every (non-local) variable's purpose. The variables can be found in Table 1.

Setup (1^Λ)

- $(pp_{\mathsf{SEAC}}, msk_{\mathsf{SEAC}}) \leftarrow \mathsf{SEAC.Setup}(1^\Lambda)$
- $(pp_{HT}, VK_{HT}) \leftarrow \mathsf{AAD.Setup}(1^\Lambda, \beta(\Lambda))$
- $(pp_{Man}, VK_{Man}) \leftarrow \mathsf{AAD.Setup}(1^\Lambda, \beta(\Lambda))$
- $(pp_{Dat}, VK_{Dat}) \leftarrow \mathsf{AAD.Setup}(1^\Lambda, \beta(\Lambda))$
- $(pp_{\mathsf{AAD}}, VK_{\mathsf{AAD}}) \leftarrow ((pp_{HT}, pp_{Man}, pp_{Dat}), (VK_{HT}, VK_{Man}, VK_{Dat}))$
- $(ISK, IVK) \leftarrow \Sigma.\mathsf{KeyGen}(1^\Lambda)$
- $(JSK, JVK) \leftarrow \Sigma.\mathsf{KeyGen}(1^\Lambda)$
- $PK \leftarrow (pp_{\mathsf{SEAC}}, pp_{\mathsf{AAD}}, VK_{\mathsf{AAD}}, IVK, JVK)$
- $MSK \leftarrow (msk_{\mathsf{SEAC}}, ISK)$
- output PK publicly, MSK to the key issuer and JSK to the judge

UserJoin $(PK, U; PK, MSK, CL)$

Table 1: The main (non-local) variables used in our construction of dSEAC.

Variable	Description
$ASet$	data structure for storing SEAC's access control data (data structures A) for precomputed search results
$DSet$	data structure for storing SEAC's precomputed partial search results (data structures D)
CL	list of user certificates
$Dat, \overline{Dat}$	AAD for storing document tuples and corresponding digest
$HT, \overline{HT}$	AAD used by SEAC and corresponding digest
ISK	key issuer's signing key
IVK	verification key corresponding to ISK
JSK	judge's signing key
JVK	verification key corresponding to JSK
lco	last completed operation
$Man, \overline{Man}$	AAD for storing management tuples and corresponding digest
MSK	master secret of dSEAC
msk_{SEAC}	master secret of SEAC
Ntf	data structure for storing notifications
pp_{AAD}	public parameters of the AADs
pp_{SEAC}	public parameters of SEAC
PK	public parameters of dSEAC
st_{server}	the server's state, consists of all data stored at the server
st_{uid}	state of user uid, consists of the last operation completed at the server and verified by the user
VK_{AAD}	verification key of the AADs
uak_{uid}	SEAC key of user uid
uk_{uid}	signing key of user uid
UL	update log; stores all completed operations
usk_{uid}	secret key of user uid, shorthand for $(uid, uk_{uid}, uak_{uid})$
uvk_{uid}	signature verification key corresponding to uk_{uid}

U: $(uk, uvk) \leftarrow \Sigma.\mathsf{KeyGen}(1^\Lambda)$

U: $\sigma \leftarrow \Sigma.\mathsf{Sign}(uk, U)$

U: send (U, uvk, σ) to key issuer

KI: pick user identifier uid

KI: $\sigma' \leftarrow \Sigma.\mathsf{Sign}(ISK, \langle uid, uvk\rangle)$

KI: $CL' \leftarrow CL \cup \{\langle uid, uvk_{uid} = uvk, \sigma'\rangle\}$

KI: $uak_{uid} \leftarrow \mathsf{SEAC.KeyGen}(pp_{\mathsf{SEAC}}, msk_{\mathsf{SEAC}}, uid, U)$

KI: publish CL' and send (uid, uak_{uid}) to user (add it to CL)

U: $usk_{uid} \leftarrow (uid, uk_{uid} = uk, uak_{uid})$

U: $st_{uid} \leftarrow \emptyset$

Init $(PK, CL, usk_{uid}, st_{uid}; PK, CL, st_{\text{server}})$

U: send "init" to the server

S: if $st_{\text{server}} \neq \perp$: send "already initialized" to user and abort protocol.

S: initialize three empty append-only authenticated dictionaries with unauthenticated data (HT, Man, Dat) that correspond to pp_{AAD} and create initial digests for them, resulting in dictionaries HT, Man, Dat with digests $\overline{HT}, \overline{Man}, \overline{Dat}$, respectively

S: create empty sets UL, Ntf

S: create empty sets $ASet$, $DSet$ for storing data structures `A` and `D`, respectively, from SEAC instances

S: send $dig = (\overline{HT}, \overline{Man}, \overline{Dat})$ to user

U: if received "already initialized:" abort the protocol

U: $\sigma \leftarrow \Sigma.\mathsf{Sign}(uk_{uid}, (0, dig, \emptyset, uvk_{uid}))$

U: $st_{uid} \leftarrow (0, dig, \emptyset, uvk_{uid}, \sigma)$

U: send st_{uid} to server

S: verify the signature in st_{uid} and that uvk_{uid} occurs in CL

S: if verification fails: abort the protocol

S: add $lco = st_{uid}$ to UL

S: $st'_{\text{server}} \leftarrow (HT, Man, Dat, UL, Ntf, ASet, DSet, lco)$

S: send "init complete" to user

Update $(PK, CL, usk_{uid}, st_{uid}, B; PK, CL, st_{\text{server}})$

U: send "update" to server

S: if another Update operation is being processed: send "concurrent operation" to user and abort protocol

S: if a temporary notification n is valid: send n to user

S: send lco from st_{server} to user

U: if received "concurrent operation:" abort protocol

U: if received notification n: verify n; if verification fails: abort protocol and raise alarm

U: verify the signature in lco

U: if verification fails: abort protocol and raise alarm

U: request append-only proofs for digests from st_{uid} and lco; let t be the first component from lco

S: send requested proofs ($\mathsf{AAD.ProveAppend}$) to user

U: verify proofs ($\mathsf{AAD.VrfyAppend}$)

U: if verification fails: abort protocol and raise alarm

U: $st_{uid} \leftarrow lco$

U: for all documents from B in random order compute (new versions of) management and data tuples as needed; check Man and Dat for old versions of the tuples, if applicable

U: compute $I \leftarrow \mathsf{SEAC.BuildIndex}(pp_{\mathsf{SEAC}}, uak_{uid}, t+1, B)$

U: $\sigma_I \leftarrow \Sigma.\mathsf{Sign}(uk_{uid}, I)$

U: send I and management and data tuples to server

S: AAD.Append computed management and data tuples and HT-entries from I to Man, Dat and HT, respectively; use the Man and Dat tuples' tid fields as their labels; obtain updated digests $\overline{Man}, \overline{Dat}, \overline{HT}$

S: send $dig = (\overline{Man}, \overline{Dat}, \overline{HT})$ to user

U: $\sigma \leftarrow \Sigma.\mathsf{Sign}(uk_{uid}, (t+1, dig, \sigma_I, uvk_{uid}))$

U: send $lco = (t+1, dig, \sigma_I, uvk_{uid}, \sigma)$ to server

S: verify lco, make sure the data from lco matches the expected data, and check that uvk_{uid} occurs in CL

S: if verification fails: send "invalid operation," roll back all changes made to Man, Dat, HT and their digests during this Update operation, and abort the protocol

S: add `A` and `D` from I to $ASet$ and $DSet$, respectively

S: add lco to UL

S: st'_{server} consists of the (potentially updated) data from st_{server}

S: send "update complete" to user

U: if received "update complete:" $st'_{uid} \leftarrow lco$

U: otherwise: abort the protocol

Search $(PK, CL, usk_{uid}, kw; PK, CL, st_{\text{server}})$

U: send "search" to server

S: if an Update operation is being processed: send "concurrent operation" to user and abort protocol

S: send lco form st_{server} to user

U: if received "concurrent operation:" abort protocol

U: verify the signature in lco and that lco's uvk_{uid} occurs in CL

U: if verification fails: abort protocol and raise alarm

U: request append-only proofs for digests from st_{uid} and lco; let t be the first component from lco

S: send requested proofs (AAD.ProveAppend) to user

U: verify proofs (AAD.VrfyAppend)

U: if verification fails: abort protocol and raise alarm

U: $st'_{uid} \leftarrow lco$

U: $query \leftarrow \mathsf{SEAC.Trpdr}(pp_{\mathsf{SEAC}}, uak_{uid}, t, kw)$

U: $\sigma \leftarrow \Sigma.\mathsf{Sign}(uk_{uid}, query)$

U: send $(query, \sigma, uvk_{uid})$ to server

S: verify the signature on $query$ and that uvk_{uid} occurs in CL

S: if verification fails: send "operation disallowed" to user and abort the protocol

S: if $query$ is malformed: send "query malformed" to user and abort the protocol

S: $rslt \leftarrow \emptyset$

S: to each searchable SEAC collection I, apply $\mathsf{SEAC.Search}(pp_{\mathsf{SEAC}}, I, query)$; add each result to $rslt$, including a proof that all collections have been considered (AAD.ProveMemb)

S: if a SEAC collection is malformed, engage in conflict resolution; a temporary or permanent notification for SEAC collection from time step t replaces the search result from the corresponding collection in *rslt*

S: send *rslt* to user

U: if received "operation disallowed" or "query malformed:" abort the protocol

U: verify the correctness and completeness of the search result (AAD.VrfyMemb, signature verification)

U: if verification fails: abort protocol and raise alarm

U: decrypt document ciphertexts from *rslt* and output resulting plaintexts

Resolve $(PK, CL, st_{\text{server}}; PK, CL, JSK)$

S: let t be the time stamp of the flawed collection

S: send the flawed SEAC collection, the signed *query* that led to the detection of the flaw, the *UL* entry for time stamp t, *lco* and append-only proofs of the digests in the *UL* entry and *lco* to the judge

J: let t' be the time stamp from *lco*

J: verify the signature in *lco* and that *lco*'s uvk_{uid} occurs in *CL*

J: if verification fails: abort protocol and raise alarm

J: compute temporary notification for time stamp t with validity t'

J: send temporary notification to server

J: verify the signatures and the append-only proofs (AAD.VrfyAppend) from the received data and that the potentially flawed collection is from time step t

J: if verification fails: abort protocol and raise alarm

J: apply SEAC.Search to the potentially flawed collection

J: if the collection is malformed: compute a permanent notification n for time stamp t; let σ_n be the signature from n

J: otherwise: abort the protocol and raise alarm

J: let *dig* be the digests from *lco*

J: $\sigma \leftarrow \Sigma.\mathsf{Sign}(JSK, (t'+1, dig, \sigma_n, JVK))$

J: send $lco = (t'+1, dig, \sigma_n, JVK, \sigma)$ and n to server

S: add n to *Ntf*

S: add *lco* to *UL*

S: st'_{server} consists of the (potentially updated) data from st_{server}

Correctness and efficiency. Our correctness notion for dynamic searchable encryption with access control requires search results to contain all documents that are accessible to the user on whose behalf search is performed and that contain the keyword, with the exception of documents introduced via broken updates, in which case a notification — a certificate of an update's brokenness — replaces the document.

In dSEAC, these requirements are met. The correctness of the underlying SEAC scheme makes sure that correctness holds for each SEAC collection individually. dSEAC makes sure that every SEAC collection is considered. dSEAC also replaces broken SEAC collections

by notifications, if a collection's brokenness is known in advance or discovered during the search process. Hence, dSEAC is correct.

The efficiency of dSEAC is heavily dependent on the performance of the underlying append-only authenticated dictionary and the efficiency of SEAC, while the efficiency of SEAC is heavily dependent on the number of keyword–(read) policy pairs, both in terms of storage and computation. Generally, if the number of keyword–policy pairs is high, the number of MA-CP-ABE decryptions during search is high on the server's side, and both, users and the server, need to perform a high number of signature verifications (for list authenticators). Likewise, during updates, the number of keyword–read policy pairs is linked to the number of MA-CP-ABE ciphertexts and signatures that a user needs to compute.

All in all, the size of the data structure `A` in SEAC linearly depends on the number of keyword–read policy pairs. In dSEAC, the equivalent of that data structure, *ASet*, has a size that is the sum of the sizes of the corresponding data structure in SEAC collections. This relation between the sizes of SEAC and corresponding dSEAC data structures also applies to *DSet* and *HT*. However, in each SEAC collection, the size of `D` linearly depends on the number of document–keyword pairs, and the size of the dictionary linearly depends on the number of keywords. The sizes of data structures *Man* and *Dat* linearly depend on the number of documents.

Security. Our dSEAC scheme as presented above satisfies our four security notions for dynamic searchable encryption schemes with access control.

Theorem 10 (informal)**.** *dSEAC provides data confidentiality, and is fork consistent, verifiable, and forward private.*

The properties follow from Lemmas 11–14, which can be found in Appendix D. Note that the proofs rely on details of our schemes presented in Appendix B, and the leakage functions presented in Appendix C.

The proofs can be briefly summarized as follows:

- Due to our simulation-based definition of *data confidentiality*, the simulator must be able to compute data structures, ciphertexts, and (hiding) signatures on data it does not know at the time these things need to be computed, so the only option for the simulator is to compute dummy data; however, at a later time, the original data becomes known. Therefore, the simulator must be able to patch the data structures, ciphertexts and hiding signatures so the dummy data matches the real data. Such patching can be achieved in the random oracle model. The ability to patch the random oracle is the core property to achieve indistinguishability of a simulated system from a real system, and thus data confidentiality.

- *Fork consistency* and *verifiability* are inherited from authenticated append-only dictionaries.

- As mentioned before, *forward privacy* is due to creation time dependence.

5 Discussion

Searchable encryption schemes often distinguish several types of users, e.g. data users and data owners [21, 22, 23]. Explicitly distinguishing users from owners makes their respective roles clearer and is a first step in the direction of fine-grained access control

with respect to write access. However, in practice, these types are often merely roles of the same user. Therefore it seems unnatural to have separate enrollment processes for the roles. Nevertheless, dSEAC allows for making different user roles explicit by means of attribute-based encryption: simply introduce role-specific attributes. For example, a user serving in the role of data owner might get attributes for document ownership and write access, whereas the user that is restricted to the role of data user might only get attributes for read access to documents and access to search results.

Many schemes that address fine-grained access control to remotely stored data via attribute-based encryption provide explicit means of user revocation [19, 21, 22, 23, 24]. A standard approach to user revocation requires re-encryption of all ciphertexts that a revoked user has access to. Thus, revoking users is very costly. In dSEAC, user revocation can be achieved by revoking user certificates. As a result, the revoked user's certificate is removed from the certificate list CL. Since users are required to include a signature under their signing key in every query they make to the server, the server can check whether a user's verification key is included in CL. If the user's verification key is not included in CL, the server does not process the user's request. For this to work properly, the server needs to fully cooperate. However, re-encryption of ciphertexts, and thus, not distributing old ciphertexts, requires the same degree of cooperation.

With our dSEAC construction, we address an open question of Löken [13], who asked whether searchable encryption with access control can be made verifiable, particularly in case of dynamic document collections, and, by extension, what price needs to be paid for dynamics and verifiability in terms of leakage. Our answer to Löken's question is that there is no significant leakage difference for dynamic and static document collections.

We note here that we have opted for a rather strong formalization of MA-CP-ABE and its security, and, as discussed, no scheme presently satisfies the notion. We have made our choice as to limit the scope of our discussion of MA-CP-ABE and provide a single concise definition of security. Technically, dSEAC can be realized and proven secure under a notion of MA-CP-ABE that is weaker than ours in terms of features and security. Particularly, in dSEAC there is no need for attribute authorities to be fully independent (c.f. Section 3.2.3), because master keys of all attribute authorities are held by the same entity, namely the key issuer. However, it must be possible for users, during search, to derive keyword-specific keys for search queries from customized authority secrets without coordinating with the key issuer. It is also sufficient to consider the cpa-security of MA-CP-ABE and security of authority key customization separately, as is done in the original work on authority key customization by Löken [13].

Acknowledgements

This work was supported by the Ministry of Culture and Science of the German State of North Rhine-Westphalia within the research program "Digital Future."

References

[1] Alderman, J., Martin, K.M., Renwick, S.L.: Multi-level access in searchable symmetric encryption. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected

Papers. Lecture Notes in Computer Science, vol. 10323, pp. 35–52. Springer (2017), https://doi.org/10.1007/978-3-319-70278-0_3

[2] Bethencourt, J., Sahai, A., Waters, B.: Ciphertext-policy attribute-based encryption. In: 2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA. pp. 321–334. IEEE Computer Society (2007), https://doi.org/10.1109/SP.2007.11

[3] Boneh, D., Crescenzo, G.D., Ostrovsky, R., Persiano, G.: Public key encryption with keyword search. In: Cachin, C., Camenisch, J. (eds.) Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3027, pp. 506–522. Springer (2004), https://doi.org/10.1007/978-3-540-24676-3_30

[4] Bösch, C., Hartel, P.H., Jonker, W., Peter, A.: A survey of provably secure searchable encryption. ACM Comput. Surv. 47(2), 18:1–18:51 (2014), http://doi.acm.org/10.1145/2636328

[5] Bost, R.: $\Sigma o \phi o \varsigma$: Forward secure searchable encryption. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1143–1154. ACM (2016), http://doi.acm.org/10.1145/2976749.2978303

[6] Cachin, C., Geisler, M.: Integrity protection for revision control. In: Abdalla, M., Pointcheval, D., Fouque, P., Vergnaud, D. (eds.) Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5536, pp. 382–399 (2009), https://doi.org/10.1007/978-3-642-01957-9_24

[7] Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: Ray, I., Li, N., Kruegel, C. (eds.) Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 668–679. ACM (2015), https://doi.org/10.1145/2810103.2813700

[8] Chase, M.: Multi-authority attribute based encryption. In: Vadhan, S.P. (ed.) Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4392, pp. 515–534. Springer (2007), https://doi.org/10.1007/978-3-540-70936-7_28

[9] Curtmola, R., Garay, J.A., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: Improved definitions and efficient constructions. Journal of Computer Security 19(5), 895–934 (2011), https://doi.org/10.3233/JCS-2011-0426

[10] Etemad, M., Küpçü, A., Papamanthou, C., Evans, D.: Efficient dynamic searchable encryption with forward privacy. PoPETs 2018(1), 5–20 (2018), https://doi.org/10.1515/popets-2018-0002

[11] Kaci, A., Bouabana-Tebibel, T.: Access control reinforcement over searchable encryption. In: Joshi, J., Bertino, E., Thuraisingham, B.M., Liu, L. (eds.) Proceedings of

the 15th IEEE International Conference on Information Reuse and Integration, IRI 2014, Redwood City, CA, USA, August 13-15, 2014. pp. 130–137. IEEE Computer Society (2014), https://doi.org/10.1109/IRI.2014.7051882

[12] Lewko, A.B., Waters, B.: Decentralizing attribute-based encryption. In: Paterson, K.G. (ed.) Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6632, pp. 568–588. Springer (2011), https://doi.org/10.1007/978-3-642-20465-4_31

[13] Löken, N.: Searchable encryption with access control. In: Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017. pp. 24:1–24:6. ACM (2017), http://doi.acm.org/10.1145/3098954.3098987

[14] Rouselakis, Y., Waters, B.: Efficient statically-secure large-universe multi-authority attribute-based encryption. In: Böhme, R., Okamoto, T. (eds.) Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8975, pp. 315–332. Springer (2015), https://doi.org/10.1007/978-3-662-47854-7_19

[15] Sahai, A., Waters, B.: Fuzzy identity-based encryption. In: Cramer, R. (ed.) Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3494, pp. 457–473. Springer (2005), https://doi.org/10.1007/11426639_27

[16] Singh, A., Liu, L.: Sharoes: A data sharing platform for outsourced enterprise storage environments. In: Alonso, G., Blakeley, J.A., Chen, A.L.P. (eds.) Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México. pp. 993–1002. IEEE Computer Society (2008), https://doi.org/10.1109/ICDE.2008.4497508

[17] Song, D.X., Wagner, D.A., Perrig, A.: Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000. pp. 44–55. IEEE Computer Society (2000), https://doi.org/10.1109/SECPRI.2000.848445

[18] Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014. The Internet Society (2014), http://www.internetsociety.org/doc/practical-dynamic-searchable-encryption-small-leakage

[19] Sun, W., Yu, S., Lou, W., Hou, Y.T., Li, H.: Protecting your right: Verifiable attribute-based keyword search with fine-grained owner-enforced search authorization in the cloud. IEEE Trans. Parallel Distrib. Syst. 27(4), 1187–1198 (2016), https://doi.org/10.1109/TPDS.2014.2355202

[20] Tomescu, A., Bhupatiraju, V., Papadopoulos, D., Papamanthou, C., Triandopoulos, N., Devadas, S.: Transparency logs via append-only authenticated dictionaries. IACR Cryptology ePrint Archive 2018, 721 (2018)

[21] Yang, K., Jia, X.: Attributed-based access control for multi-authority systems in cloud storage. In: 2012 IEEE 32nd International Conference on Distributed Computing Systems, Macau, China, June 18-21, 2012. pp. 536–545. IEEE Computer Society (2012), https://doi.org/10.1109/ICDCS.2012.42

[22] Yang, K., Jia, X., Ren, K.: Attribute-based fine-grained access control with efficient revocation in cloud storage systems. In: Chen, K., Xie, Q., Qiu, W., Li, N., Tzeng, W. (eds.) 8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013. pp. 523–528. ACM (2013), http://doi.acm.org/10.1145/2484313.2484383

[23] Yang, K., Jia, X., Ren, K., Zhang, B., Xie, R.: DAC-MACS: effective data access control for multiauthority cloud storage systems. IEEE Trans. Information Forensics and Security 8(11), 1790–1801 (2013), https://doi.org/10.1109/TIFS.2013.2279531

[24] Yu, S., Wang, C., Ren, K., Lou, W.: Achieving secure, scalable, and fine-grained data access control in cloud computing. In: INFOCOM 2010. 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 15-19 March 2010, San Diego, CA, USA. pp. 534–542. IEEE (2010), https://doi.org/10.1109/INFCOM.2010.5462174

[25] Zhang, Y., Katz, J., Papamanthou, C.: All your queries are belong to us: The power of file-injection attacks on searchable encryption. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 707–720. USENIX Association (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/zhang

[26] Zhao, F., Nishide, T., Sakurai, K.: Realizing fine-grained and flexible access control to outsourced data with attribute-based cryptosystems. In: Bao, F., Weng, J. (eds.) Information Security Practice and Experience - 7th International Conference, ISPEC 2011, Guangzhou, China, May 30 - June 1, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6672, pp. 83–97. Springer (2011), https://doi.org/10.1007/978-3-642-21031-0_7

[27] Zheng, Q., Xu, S., Ateniese, G.: VABKS: verifiable attribute-based keyword search over outsourced encrypted data. In: 2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014. pp. 522–530. IEEE (2014), https://doi.org/10.1109/INFOCOM.2014.6847976

A Comparisons in ABE policies

The policies used in attribute-based encryption (ABE), including MA-CP-ABE, typically do not allow comparisons of numeric values of attributes to be expresses directly. In particular, this is true for ABE schemes that use monotone span programs to express policies. For example, if there are attributes age16, age18, and age21, a property like age $\geq$ 18 cannot be expressed as a single statement; it must be decomposed into the policy age18 $\vee$ age21. While this is feasible if the number of options is low, it becomes impractical if the range of options is huge.

For example, in our dSEAC scheme, we would like to have a comparison *now* $\geq$ creation_time, where both, *now* and creation_time are discrete time stamps. In this case, the above idea is still somewhat applicable, albeit, we must change our view on the matter:

if we see *now* and creation_time as bitstrings, we can apply the above idea in a bit by bit fashion, starting with the most significant bits. We find the first bit position in which the strings differ, and use the respective bits to determine which of the numbers represented by the bitstrings is the greater one. Upon reaching a conclusion, no further comparison is required.

For us to be able to express this idea in terms of an ABE policy, we represent the bitstrings as sets of attributes, one attribute for each bit position in the bitstring. This technique is straightforward to implement as a policy, and achieves an exponential speedup in comparison to the original idea of enumerating all potential values that may be compared.

B Instantiating dSEAC

We now provide a more detailed description of the dSEAC scheme. Due to dSEAC's reliance on and non-blackbox use of SEAC [13], we need to discuss encrypted linked lists first, and in more detail than in Section 3.2.4, as (encrypted) lists are an integral building block for SEAC. Particularly, we discuss the lists as used in dSEAC, i.e. we also consider our own extensions upon SEAC, c.f. Sections 4.1.1–4.1.4.

SEAC and dSEAC use encrypted linked lists to store precomputed search results, as is done by Curtmola et al. in their SSE-1 scheme [9]. An encrypted linked list is a linked list that is symmetrically encrypted in a node by node fashion. With each list node, a pointer to the successor node and the successor node's decryption key is stored. This approach requires special treatment of pointers to and keys of list heads. Such pointers and keys are stored in a separate location.

Particularly, in dSEAC, we have two types of encrypted linked lists, called `A` and `D` lists. The `D` lists store — for each keyword, creation time, and (read) policy — pointers to documents that are associated to the given policy and contain the given keyword. See Figure 9 for a concise overview of a `D` list node in dSEAC.

$$\mathsf{Sym.Enc}\left(k_{\text{node}}, \langle p_{\text{successor}}, k_{\text{successor}}, \mathit{tid}, v_{mt}, v_{dt}\rangle\right)$$

Figure 9: The internals of a `D` list node. The entry is symmetrically encrypted under key k_{node}. The plaintext consists of a pointer and decryption key for the node's successor $(p_{\text{successor}}, k_{\text{successor}})$ and a pointer to a document. Particularly, the document pointer consist of the tuple identifier *tid* and version numbers v_{mt} and v_{dt} of the document version's management and data tuples, respectively.

Pointers to and keys of `D` lists' heads are stored in `A` lists for the relevant keywords and creation times. There is one `A` list for each keyword and creation time. Pointers and keys to `D` list heads are encrypted using attribute-based encryption. Since `A` lists are encrypted linked lists, their general structure is similar to the structure of `D` lists, as can be seen in Figure 10, especially Figure 10a. The list node also stores the policy $\mathbb{A}$ and a list authenticator of the `D` list it points to. The list authenticator $\sigma^{(\texttt{D list})}_{kw,t,\mathbb{A}}$ is a *hiding* signature on the set of document pointers from the `D` list under the signature verification key of the list's creator. Particularly, the list node displayed in the figure points to a `D` list created at time t that is induced by some keyword kw and read policy $\mathbb{A}$. As shown in Figure 10c, dSEAC stores the pointer to and key of the `D` list head together with the open value $d^{(\texttt{D list})}_{kw,t,\mathbb{A}}$ of the `D` list's authenticator.

$$\mathsf{Sym.Enc}\left(k_{\text{node}}, \left\langle p_{\text{successor}}, k_{\text{successor}}, \mathbb{A}, \sigma^{(\mathtt{D}\ \text{list})}_{kw,t,\mathbb{A}}, \mathsf{ABE.Enc}\left(pp, \mathbb{A}^{+}, plaintext\right)\right\rangle\right)$$

(a) General structure of an `A` list node. The node stores a successor pointer and key $(p_{\text{successor}}, k_{\text{successor}})$, a policy $\mathbb{A}$, a list authenticator $\sigma^{(\mathtt{D}\ \text{list})}_{kw,t,\mathbb{A}}$, and an ABE ciphertext.

$$\begin{aligned}\mathbb{A}^{+} = \mathbb{A}\ \&\text{“}&\textit{Time}{:}\geq t\text{”}\\ \&\text{“}&\textit{Sys}{:}\text{search”}\\ \&\text{“}&\textit{Srch}{:}f_B\left(k_B, kw\right)\text{”}\end{aligned}$$

(b) The extended policy for `A` list nodes' internal ABE ciphertext.

$$plaintext = \left\langle p^{(\mathtt{D}\ \text{list})}_{kw,t,\mathbb{A}}, k^{(\mathtt{D}\ \text{list})}_{kw,t,\mathbb{A}}, d^{(\mathtt{D}\ \text{list})}_{kw,t,\mathbb{A}}\right\rangle$$

(c) The plaintext that corresponds to `A` list nodes' internal ABE ciphertext. It consists of a pointer to a `D` list, the list head's decryption key, and the list authenticator's open value.

Figure 10: The internals of an `A` list node, particularly for keyword kw and creation time t.

Figure 10b shows the access policy used for encrypting information on the `D` list. The policy is an extension of the access policy $\mathbb{A}$ of documents referred to in the `D` list; it is extended by a policy based on the current time (so only future search queries can access the `D` list), the search attribute (c.f. Section 3.2.4), and an attribute derived from the keyword (that serves as a binder term).

Pointers to and keys of `A` lists' heads are stored in an authenticated dictionary with unauthenticated data HT in an encrypted form. Figure 11a shows the structure of an HT entry, which consist of a label, an authenticated tuple comprised of a signature verification key uvk_{uid} of the tuple's creator and a list authenticator $\sigma^{(\mathtt{A}\ \text{list})}_{kw,t}$, and an unauthenticated encrypted value. The list authenticator stored in the HT entry is a *hiding* signature on the access policies and list authenticators contained in the `A` list that the HT entry points to; the signature is under the signature verification key of the HT entry's creator.

As can be seen from Figures 11b and 11c, respectively, the label and key depend on the keyword in two ways. The actual label and key are derived via two keyed functions F, F', respectively. Bost defines F and F' to be hash functions [5]. The functions' keys are derived from the keyword (so only enrolled users can form these tuples and make search queries), while the arguments are derived from a keyword-specific seed by repeatedly applying the inverse of a trapdoor permutation π (so old search queries do not yield results from the update that introduced the entry into HT). In accordance with Bost, c.f. Section 5.5 of [5], we suggest the seeds to be derived deterministically from the keyword via some keyed function f_S, i.e. $seed_{kw} = f_S(k_S, kw)$.

The plaintext pointer to an `A` list is shown in Figure 11d. As can be seen, alongside the pointer to and key of the list head $(p^{(\mathtt{A}\ \text{list})}_{kw,t}, k^{\mathtt{A}\ \text{list})}_{kw,t})$, the HT entry stores its creation time t, and an open value $d^{(\mathtt{A}\ \text{list})}_{kw,t}$ for the list authenticator $\sigma^{(\mathtt{A}\ \text{list})}_{kw,t}$.

For security, nodes of `A` and `D` lists are stored at random locations in two memory arrays of sufficient size; one array for each type of list. Free memory cells in the arrays are filled with dummy entries that are indistinguishable from not yet decrypted list nodes. We further discuss the effect of dummy entries on the sizes of data structures `A` and `D` below.

$$\left\langle label, \left\langle uvk_{uid}, \sigma_{kw,t}^{(\texttt{A list})} \right\rangle, key \oplus plaintext \right\rangle$$

(a) General structure of HT entries: a label and an encrypted value.

$$label = F\left(f_L\left(k_L, kw\right), \pi^{-t}\left(SK_\pi, seed_{kw}\right)\right)$$

(b) Structure of HT entries' label component: a keyed function F applied to a keyword dependent key and a keyword and time dependent argument.

$$key = F'\left(f_K\left(k_K, kw\right), \pi^{-t}\left(SK_\pi, seed_{kw}\right)\right)$$

(c) Structure of HT entries' key component: a keyed public function F' applied to a keyword dependent key and a keyword and time dependent argument.

$$plaintext = \left\langle t, p_{kw,t}^{(\texttt{A list})}, k_{kw,t}^{(\texttt{A list})}, d_{kw,t}^{(\texttt{A list})} \right\rangle$$

(d) Structure of HT entries' plaintext component: the creation time of the entry, a pointer to the head of some `A` list, the decryption key for that list head, and the open value for the `A` list's authenticator.

Figure 11: The internals of a dictionary entry in HT, particularly for keyword kw and creation time t.

$$\left(\begin{array}{c} \pi^{-t}(SK_\pi, seed_{kw}), f_L(k_L, kw), f_K(k_K, kw), \\ R^{(uid)} \cup \{usk_{uid,\text{“}Time:=\ t\text{”}}, usk_{uid,\text{“}Sys:\text{search}\text{”}}, usk_{uid,\text{“}Srch:f_B(k_B,\,kw)\text{”}}\} \end{array}\right)$$

Figure 12: A dSEAC search query for keyword kw at time t by user uid with access rights $R^{(uid)}$, i.e. attributes of the users uid managed by authority Usr.

How to search. With the lists nodes and HT entries as above, a search query is as shown in Figure 12. Using the value of the trapdoor permutation π^{-1} and the values of f_L and f_K, respectively, the server can recompute all labels and decryption keys for HT entries, even for previous time steps, while the attributes provided as part of the query allow for the decryption of ABE ciphertexts from `A` list nodes if uid's access rights satisfy the nodes' access policies. Since t is the number of updates that have occurred, the server knows t, and thus, how many labels to reconstruct.

For each of the computed labels, the server includes a proof of (non-)membership of the label in HT in the search result. If there is an HT entry for some label, the server also includes the authenticated data that HT associates to the label and the corresponding unauthenticated open value in the search result, while the other unauthenticated data is used to access some `A` list. From the `A` list, the server includes all plaintext access policies and list authenticators in the search result, and attempts to decrypt the ABE ciphertexts from the `A` list using the attributes from the search query. If decryption succeeds, the server accesses the respective `D` lists, and includes the `D` lists' document pointers, as well as the management and data tuples being pointed to, in the search result.

When receiving such a search result, the user can use the (non-)membership proofs to check that the server has considered all updates, and what updates contain the searched keyword. If an update contains the keyword, then the search result also contains an `A` list authenticator and all the signed data required to verify the signature. That additional data can also be used to determine that the server has considered all access policies from the respective `A` list, that the server has included all document pointers from all `D` lists that the server was supposed to access, as well as the respective documents' management and data tuples.

If for some update there is a temporary or permanent notification, the server includes the notification instead of any data from the update in the search result, and the user verifies the notification rather than the (missing) (non-)membership proof.

Motivation for tuple structure. It may seem odd that dSEAC uses two *protection mechanisms that are time dependent,* namely time dependent access policies in A list nodes and time dependent HT labels and keys in HT entries. One may ask whether one of these is sufficient, or could be adapted to the other data structure, without loss of efficiency or security.

In order for dSEAC to be forward private, two HT entries for the same keyword must have different labels and decryption keys. It is not clear how different labels for the same keyword can be achieved using access policies. It must also be noted that the label derivation process must be deterministic as to avoid synchronization between users and owners. A lack of such synchronization is particularly desirable because some users and owners may be dishonest. Decryption for HT entries could be based on policies (so HT entries would contain an ABE ciphertext rather than a symmetric ciphertext), but achieving time dependence from trapdoor permutations is more efficient than achieving it from ABE, particularly if ABE is not used for any other purpose. Additionally, the policy would still need to consider the keyword, which may in turn be a threat to forward privacy.

On the other hand, achieving time dependence in A list entries from trapdoor permutations does not work, because it either detaches the ABE policy from time constraints completely, or there is an attribute in the policy that is computed using the trapdoor permutation. In the latter case, the search query must contain such attributes for all previous time steps, or the server needs to be able to compute such attributes. Neither option is attractive: the first option is inefficient and does not use the advantage that the trapdoor permutation provides (enabling the server to compute labels and keys), the second option is outright insecure (not forward private) — the server can use the value of the trapdoor permutation from one search query and user's attributes from an earlier query for the same keyword to perform search for the new time step, but with access rights from the earlier query. This type of attack also arises if the ABE policy is detached from time constraints completely.

A second question is why we use hiding signatures for our list authenticator. This is due to bandwidth efficiency and security. With our data structures, verifying a list authenticator requires no knowledge about internal details of the data structure, e.g. the concrete values of successor pointers and list keys. This is the advantage of hiding signatures when compared to applying the encrypt–then–sign principle, which would result in similar security guarantees.

On the other hand, all relevant data is authenticated. Every verifier can verify all signatures she receives, without necessarily knowing the underlying messages. This is important, because, despite having incomplete information, a verifier, during search, can still check that the server has provided the correct and complete search result. Particularly, even a verifier whose access rights do not satisfy any access policy from an A list can still check that the server has considered all access policies from the A list when computing the search result. This even holds in a setting in which the server can corrupt users and, thus, may try to omit some results and replace the original signatures with signatures created using the corrupt user's singing key.

Data structure sizes and padding. In the main body of our text, we have explained that the number of (useful) entries of data structures HT, A and D of a SEAC instance

computed from a document batch B depends on certain properties of B. Particularly, the number of useful entries in HT linearly depends on the number of distinct keywords in B, the number of useful entries in `A` linearly depends on the number of keyword–access policy pairs in B, and the number of useful `D` entries linearly depends on the number of keyword–document pairs in B.

Both, Curtmola et al. [9] and Bost [5], use data structures similar to those we use. Particularly, the concept of a dictionary for fast access is present in both works, while encrypted linked lists are only present in Curtmola et al.'s work. While Curtmola et al. suggest to pad all data structures with dummy entries that are indistinguishable from useful entries, Bost does not pad the data structures. Since our implementation of the dictionary HT is closer to Bost's, we choose to apply no padding to HT, but follow Curtmola et al.'s suggestions for padding `A` and `D`.

Let $\delta(B)$ be the number of distinct keywords in B, let $\alpha(B)$ be the number of distinct read access policies in B, let m be a lower bound on the length of keywords,[6] and let $\#(B)$ be the total length of B. Applying the padding suggestions, an update for B computes $\delta(B)$ entries for dictionary HT, $\delta(B) \cdot \alpha(B)$ entries for `A`, and $\#(B)/m$ entries for `D`.

Note that this is only the padding of the data structures themselves; within each data structure, there is some need for additional padding, so for each data structure, each entry has the same length. However, the lengths only need to be the same for each data structure and SEAC instance individually, so `A` entries computed from different document batches may differ in length. Size differences may be caused by read access policies (complicated policies have larger representations than simple policies), as well as the sizes of data structures entries point to (i.e. the lengths of pointers may vary).

C Leakage in dSEAC

In this section we describe and discuss the leakage incurred in dSEAC. Towards the end of this section, we interpret dSEAC's leakage in the leakage classification framework of Cash et al..

C.1 dSEAC's leakage functions

As stated in Section 3.1, dSEAC has seven leakage functions that are related to oracle calls of an adversary in the data confidentiality security experiments for dynamic searchable encryption with access control. Particularly, we have leakage functions for enrolling honest users, enrolling corrupt users, corrupting honest users, having honest users perform server initialization, perform an update of the searchable document collection, and perform search, as well as for the adversary interacting with the judge during conflict resolution.

Joining and corrupting users. Whenever a honest user joins the system, the only thing an adversary can observe is the user's certificate, as published by the key issuer. Hence, $\mathcal{L}_{\mathrm{HJ}}(U) = uid$, where U is a set of access rights, and uid is a user identifier.

For dSEAC, in terms of leakage, it does not matter whether the adversary corrupts a once-honest user or enrolls as a dishonest user. In either case, the adversary learns the user's identifier uid and her attributes U_{uid} from the user's secret key. Due to learning the user's key, the adversary learns all plaintext documents doc that can be accessed by the user. The adversary also learns relations between the document and its internal

[6] As an example, for English language documents, such a lower bound would be 4 bytes/characters; shorter words are too common.

identifiers, denoted $id(doc)$. An example of such an identifier is the tuple identifier tid of the management and data tuples that actually store the document. Additionally, the adversary learns all pairs of keywords kw and read policies $\mathbb{A}_{\text{read}}(doc)$ that occur in any document batch B_i (with $doc \in B_i$). Learning the keywords includes learning their internal identifiers, denoted $id(kw)$, such as the images of the keywords under pseudorandom functions (used extensively for HT labels, keys, and seeds, as well as for including keywords in access policies). Therefore,

$$\begin{aligned}&\mathcal{L}_{\text{UC}}(uid) = \\ &\mathcal{L}_{\text{CJ}}() = \\ &\quad\begin{pmatrix}(uid, U_{uid}),\\ \{doc, id(doc) : doc \in B_i \wedge U_{uid} \text{ satisfies } \mathbb{A}_{\text{read}}(doc)\},\\ \{kw, id(kw), \mathbb{A}_{\text{read}}(doc) : doc \in B_i \wedge kw \in doc\}\end{pmatrix}.\end{aligned}$$

Users interacting with the server During server initialization, the adversary learns nothing but which user performs the initialization. Therefore, $\mathcal{L}_{\text{Ini}}(uid) = uid$.

When performing search, the adversary learns which user performs search, what attributes that user has, and what keyword that user searches for. That information can be extracted directly from the user's search query. However, the keyword kw is not learned directly. Instead, the query contains the keyword's identifiers $id(kw)$. From performing the search procedure itself, the adversary learns the read policies of documents that contain the searched keywords, as well as the times at which documents with the given read policies have been added to the document collection. That information is stored in the `A` data structure. Additionally, the adversary learns the identifiers $id(doc)$ of documents that contain the searched keyword and that are accessible to the user, as the management and data tuples for that identifiers make up a substantial part of the search result. Thus,

$$\begin{aligned}&\mathcal{L}_{\text{Sea}}(uid, kw) = \\ &\quad\begin{pmatrix}(uid, U_{uid}), id(kw),\\ \{(\mathbb{A}_{\text{read}}(doc), t) : doc \in B_t \wedge kw \in doc\},\\ \{id(doc) : doc \in B_t \wedge U_{uid} \text{ satisfies } \mathbb{A}_{\text{read}}(doc) \wedge kw \in doc\}\end{pmatrix}.\end{aligned}$$

During updates, the adversary learns which user performs the update, e.g. from signatures, and what attributes that user has (from the management and data tuples the user updates). From the size of the data structures received during the update process, the adversary can determine upper bounds on the number of keywords (HT) the cumulated size of pre-computed search results (`D`). For every document doc from the batch B of documents that the user performs the update on, the adversary learns the bit-length $\#doc$ of the document, its internal identifiers and its read, write and ownership policies $\mathbb{A}_{\text{read}}(doc)$, $\mathbb{A}_{\text{write}}(doc)$ and $\mathbb{A}_{\text{own}}(doc)$, respectively. This information is stored in the management and data tuple representing the document. If the adversary has corrupted a user, the adversary leans plaintexts of added documents that any corrupted user has access to. Additionally, if the adversary has corrupted any user, it learns all keywords, their identifiers and the read policies of documents that are associated with a given keyword, i.e. the information that the adversary obtains on SEAC collections stored at the server at the time a user is

corrupted is also learnt for every SEAC collection created in any future update. Hence,

$$\mathcal{L}_{\mathrm{Upd}}(uid, B) = \begin{pmatrix} (uid, U_{uid}), \\ \text{number of distinct keywords in } B, \\ \text{upper bound on cumulative size of precomputed search results}, \\ \{id(doc), \#doc, \mathbb{A}_{\mathrm{read}}(doc), \mathbb{A}_{\mathrm{write}}(doc), \mathbb{A}_{\mathrm{own}}(doc) : doc \in B\}, \\ \left\{doc, id(doc) : doc \in B \wedge \exists uid' \text{ corrupt} \wedge U_{uid'} \text{ satisfies } \mathbb{A}_{\mathrm{read}}(doc)\right\}, \\ \left\{kw, id(kw), \mathbb{A}_{\mathrm{read}}(doc) : doc \in B \wedge kw \in doc \wedge \exists uid' \text{ corrupt}\right\} \end{pmatrix},$$

where "$\exists uid'$ corrupt" denotes the condition of a corrupt user with identifier uid'.

Resolution of conflicts. When engaging in conflict resolution, the adversary does not learn anything new. Therefore, $\mathcal{L}_{\mathrm{Res}}() = \emptyset$.

C.2 Leakage classification

Cash et al. have established a framework to compare the leakage that occurs in searchable encryption systems [7]. Their framework categorizes searchable encryption schemes into four hierarchically ordered leakage classes L_1–L_4.

- L_1 schemes initially only leak some size information, e.g. ciphertext lengths, upper bounds on keyword numbers, and trivial upper bounds on pre-computed results. Only when a search query is being processed, the searched keyword's access pattern, i.e. the identifiers of documents containing the keyword, is leaked.
- L_2 schemes leak essentially the same information as L_1 schemes, but the adversary learns all keywords' access patterns *immediately,* not only when a keyword is searched.
- L_3 schemes, in addition to what is leaked by L_2 schemes, also leak the order in which keywords occur in documents.
- L_4 schemes deterministically substitute keywords in the plaintext documents, so adversaries learn keywords' order and occurrence counts immediately.

It is clear that L_1 schemes leak the least information, while L_4 schemes leak essentially all information, except for keyword plaintexts. Unfortunately, the leakage classification of Cash et al. does not consider access control and user corruption. Still, we sort our dSEAC scheme into Cash et al.'s s leakage hierarchy, but ignore dSEAC's access control for the moment.

Given our discussion of leakage above, and our padding of data structures, it is clear that, as long as no user is corrupt, dSEAC's leakage can be classified as L_1. This property is inherited from the use of encrypted linked lists, as observed by Cash et al., c.f. Section 2.1 of [7]. As soon as a user gets corrupted, the leakage increases to level L_2. This is because the corrupted user's key allows the adversary to recover information on keyword's access patterns from `A` lists and, depending on the corrupted user's access rights, from `D` lists.

One may criticize the leakage of access policies that are not satisfied by the attributes from a search query. This is a particular problem if there is essentially a one-to-one relation between documents and access policies. However, for rather general classes of access policies, e.g. monotone policies, such leakage may be unavoidable if users are

supposed to be able to verify the correctness of search results, which includes checking that some potential search results have legitimately excluded from the reported result. On the other hand, for more restricted classes of access policies, e.g. a totally ordered set of access policies, leakage of access policies not satisfied by access rights may be avoided. For example, the technique of Alderman et al. [1] may be adapted to the verifiable setting while avoiding leakage of access policies not satisfied by the access rights of the user on whose behalf search is performed.

D Security proofs

In this section, we give proofs of dSEAC's security, as claimed in Section 4.2.

D.1 Data confidentiality

Lemma 11. *If dSEAC is instantiated with a cpa-akc-secure MA-CP-ABE scheme with authority key customization, a cpa-secure symmetric encryption scheme, a euf-cma secure (hiding) signature scheme (with delayed verifiability), and a secure authenticated append-only dictionary, in the random oracle model, dSEAC provides data confidentiality relative to leakage functions $\mathcal{L}_{HJ}$, $\mathcal{L}_{CJ}$, $\mathcal{L}_{UC}$, $\mathcal{L}_{Ini}$, $\mathcal{L}_{Upd}$, $\mathcal{L}_{Sea}$ and $\mathcal{L}_{Res}$ as presented in Appendix C.*

Proof. We first discuss the use of random oracles within our simulation, before we turn to the simulation itself.

Hiding signatures in the random oracle model. As mentioned in Section 3.2.1, signatures satisfying the hiding property can be constructed from digital signature schemes and commitment schemes employing a commit–then–sign paradigm. Let H be a random oracle. Then, we model the commitment scheme as follows:

Commit (m): sample $r \leftarrow_{\$} \{0,1\}^{\Lambda}$, compute $c \leftarrow H(\langle r, m\rangle)$ and $d = (r, m)$, and output (c, d).

Verify (c, d): if $c = H(\langle r, m)$ output *valid*, otherwise output *invalid*.

The value c is a commitment on the message m, while d is the corresponding decommit value. Randomness r is used so an adversary cannot guess the input to the random oracle, and thus assures that the hiding property holds, even if the adversary can guess m.

dSEAC uses hiding signatures as list authenticators. A list is singed by committing to the list, singing the commitment, and storing value r in an encrypted form together with the pointers to the list, i.e. the variable *rnd* from HT entries and `A` list nodes takes the value r. The list authenticators consists of the commitment and its signature. As a result, the decommit value can be reconstructed upon decryption of the signed list.

The purpose of hiding signatures from random oracles. Hiding signatures in the above sense are actually not committing. The use of the random oracle allows the simulator to output list authenticators on lists not known to the simulator during signature creation. If the list becomes know to the simulator at a later point in time, the simulator can patch the random oracle so the list authenticator is valid for the given list.

In the data confidentiality experiment, the adversary chooses document batches that are to be added to the searchable document collection. Since the simulator, at creation time, only obtains leakage from these document batches, the adversary has much more

knowledge about the batches than the simulator. The non-committing nature of our hiding signatures guarantee that the adversary cannot leverage its superior knowledge, and force the simulator to commit to data it does not know. Such a possibility would enable the adversary to distinguish between the real and simulated setups of dSEAC. Consider, for example, our example document collection, c.f. Figure 1. While the data structures produced in the real setup would look (more or less) as shown in the figure, the simulator does not even know the lengths of the D lists to produce.

Non-committing encryption. Asymmetric encryption is naturally committing, because otherwise correctness of the encryption scheme does not hold. However, combining asymmetric and symmetric encryption in a hybrid approach, and modelling the symmetric part as a random oracle, allows for non-committing encryption.

Non-committing asymmetric encryption is required for our simulator to successfully simulate document ciphertexts. Non-committing symmetric encryption is required for our simulator to successfully simulate encrypted linked lists. These requirements are due to the asymmetry in knowledge between the simulator and the adversary, as explained above in the case of hiding signatures.

More random oracles. In addition to the random oracles used for achieving hiding signatures and non-committing encryption, we also model other functions as random oracles. Specifically, the functions f_L, f_K, f_B, and f_S, as well as the trapdoor permutation π, c.f. Appendix B, are modelled as random oracles.

The functions are keyed, and the (secret) keys are given only to users. As a result, f_L, f_K, f_B, f_S, and π^{-1} can only be evaluated by users. In terms of our simulation, this means that the adversary can only make meaningful queries to these oracles after it has corrupted some user.

Simulation. We now turn to the simulator itself. Upon startup, the simulator runs $\mathsf{dSEAC.Setup}(1^\Lambda)$, but keeps all secret keys (MSK, ISK, JSK) to itself. The simulator reacts to leakage given to it as follows.

Upon receiving $\mathcal{L}_{\mathrm{HJ}}(U)$, the simulator picks a signing–signature verification key pair for the user, and then computes a certificate for that verification key. The simulator then publishes the certificate, and stores the user's singing key for future use.

Upon receiving $\mathcal{L}_{\mathrm{CJ}}()$, the simulator executes protocol UserJoin, with the adversary playing the corrupt user's part of the protocol. The simulator also patches its random oracles used for non-committing encryption, so data tuples point to correct document plaintexts. All oracles are patched to create D lists grouping together documents that the corrupt user has access to, to create all A lists, and for HT entries to point to these A lists, if there is an A list for the given keyword and time stamp.

Upon receiving $\mathcal{L}_{\mathrm{UC}}(uid)$, the simulator computes the part of the user's key and state that have not been computed before.[7] Oracles are patched as in the case of joining a corrupt user. Finally, the user's key and state are given to the adversary.

Upon receiving $\mathcal{L}_{\mathrm{Ini}}(uid)$, the simulator runs protocol Init, with the adversary playing the server's part. Note that knowledge of the user's signing key is sufficient for the simulator to run the protocol. The user's state is stored by the simulator for future use.

[7] The signing key has already been computed, other parts of the key may have been computed if the user has performed an update or search operation. Also, depending on previous operations of the user, the user's state may or may not be empty.

Upon receiving $\mathcal{L}_{\text{Sea}}(uid, kw)$, the simulator computes the part of the user's key and state that have not been computed before, c.f. Footnote 7. The simulator follows protocol Search, and patches its oracles so HT entries exist for the searched keyword and all relevant time stamps, and these HT entries point to `A` lists on access policies from the leakage. Oracles are also patched for these `A` lists to point to `D` lists, if applicable, that point to documents, i.e. management and data tuples, from the leakage. The user's key and (post–update) state are kept for future use.

Upon receiving $\mathcal{L}_{\text{Upd}}(uid, B)$, the simulator computes the part of the user's key and state that have not been computed before, c.f. Footnote 7. Then, the simulator interacts with the adversary as prescribed by protocol Update. However, the simulator creates dummy entries for HT, `A` and `D`, as well as dummy data tuples (using non-committing encryption, resulting in data tuples of correct length); management tuples can be computed as in the protocol. If there are corrupt users, the simulator uses the resulting additional leakage to immediately patch its oracles as it does upon user corruption.

Upon receiving $\mathcal{L}_{\text{Res}}()$, the simulator executes the judge's part of protocol Resolve, with the adversary playing the server's part.

Indistinguishability. The use of cpa-akc-secure MA-CP-ABE and symmetric encryption schemes implies that the adversary cannot distinguish the simulator's dummy ciphertexts (in `A` lists, `D` lists, HT entries, and data tuples) from ciphertexts that would be produced in the real setting $\mathbf{Real}_{\mathsf{dSEAC},\mathcal{A}}(\Lambda)$ as long as the adversary $\mathcal{A}$ is unable to decrypt these dummy ciphertexts. However, as soon as the adversary becomes able to decrypt, the simulator obtains leakage that it uses to patch its oracles, and as a result, the adversary can, again, not distinguish between the real setting and the simulated setting. This is possible due to non-committing encryption. Regarding list authenticators, the hiding notion supports ciphertext indistinguishability, because we can sign an unknown plaintext, publish the signature alongside the corresponding (dummy) ciphertext, and, as soon as the plaintext becomes known, patch our random oracles so the signature is a valid signature on the plaintext.

In either case, patching an oracle may fail if patching the oracle requires redefining a function value of the random oracle. However, such events happen only with probabilities negligible in the security parameter Λ. □

D.2 Fork consistency and verifiability

We note that our dSEAC scheme allows the server to withhold the latest updates that have occurred from user that have not seen these updates yet. Such behavior cannot be prevented, because the server can simply refuse to work. Therefore, such behavior cannot be seen as a breach of verifiability. However, if the server withholds the latest updates from a user during an update operation, but accepts the user's update, then a fork occurs.

Lemma 12. *If dSEAC is instantiated with a euf-cma secure signature scheme and a fork consistent authenticated append-only dictionary, then dSEAC is fork consistent.*

Proof. In order for an adversary to break dSEAC's fork consistency, the adversary needs to compute three last completed operations lco_i, lco'_i, lco_j such that one (honest) user has successfully verified (or performed) lco_i, another (honest) user has successfully verified (or performed) lco'_i, and afterwards, both users have successfully verified lco_j, where lco_i and lco'_i contain the same timestamp.

The last completed operations take the following forms.

- $lco_i = (i, dig_i, ch_i, vk_i, \sigma_i)$,
- $lco'_i = (i, dig'_i, ch'_i, vk'_i, \sigma'_i)$, and
- $lco_i = (j, dig_j, ch_j, vk_j, \sigma_j)$,

where $(\overline{HT_i}, \overline{Man_i}, \overline{Dat_i}) = dig_i \neq dig'_i = (\overline{HT'_i}, \overline{Man'_i}, \overline{Dat'_i})$. Hence, we have $\overline{HT} \neq \overline{HT'}$, or $\overline{Man} \neq \overline{Man'}$, or $\overline{Dat} \neq \overline{Dat'}$.

In order for the individual operations to be verifiable, they must at least satisfy that the σ component of a tuple is a valid signature on the first four components of the respective tuple, using the tuple's vk component as signature verification key. Assuming neither signature has been successfully forged, in order for lco_j to be successfully verified, the adversary has to produce proofs $\pi_i = (\pi_{i,HT}, \pi_{i,Man}, \pi_{i,Dat})$ and $\pi'_i = (\pi'_{i,HT}, \pi'_{i,Man}, \pi'_{i,Dat})$ such that the following statements are true:

- $\mathsf{AAD.VrfyAppend}(VK, \overline{HT_i}, \overline{HT_j}, i, j, \pi_{i,HT}) = valid$,
- $\mathsf{AAD.VrfyAppend}(VK, \overline{HT'_i}, \overline{HT_j}, i, j, \pi'_{i,HT}) = valid$,
- $\mathsf{AAD.VrfyAppend}(VK, \overline{Man_i}, \overline{Man_j}, i, j, \pi_{i,Man}) = valid$,
- $\mathsf{AAD.VrfyAppend}(VK, \overline{Man'_i}, \overline{Man_j}, i, j, \pi'_{i,Man}) = valid$,
- $\mathsf{AAD.VrfyAppend}(VK, \overline{Dat_i}, \overline{Dat_j}, i, j, \pi_{i,Dat}) = valid$, and
- $\mathsf{AAD.VrfyAppend}(VK, \overline{Dat'_i}, \overline{Dat_j}, i, j, \pi'_{i,Dat}) = valid$,

where VK is the AAD verification key shared between the various dictionaries. However, if the adversary has computed proofs that make all of the above statements true, then the adversary has broken fork consistency for at least one of the dictionaries HT, Man, or Dat, c.f. Definition 6. □

Lemma 13. *If dSEAC is instantiated with a euf-cma secure signature scheme and a membership and append-only-secure authenticated append-only dictionary, the dSEAC is verifiable.*

Proof. Assume the server to provide user with a false search result, i.e. at least one SEAC instance was skipped entirely during search (without being replaced by a notification), or not all D lists from a SEAC instance accessible using the user's access rights were accessed, or not all document ciphertext pointed to in these D lists are present in the search result.

- A server capable of skipping SEAC instances must be able to break append-only-security of the authenticated dictionaries used in dSEAC.
- A server that does not access all D list in a SEAC instance accessible using the user's access rights is capable of forging list authenticators (for the A list that pointes to the D list).
- A server that does not include all documents an accessed D list points to in a search result can forge list authenticators (for that D list).
- As an alternatively to forging list authenticators, the server may exploit maliciously created SEAC instances that contain list authenticators on partial lists. However, exploiting such behavior is acceptable for verifiability, although a maliciously created SEAC instance should be reported to the judge.

It must be noted that an adversarial server that colludes with corrupt users cannot exploit the exception from the last item of the above list. This is because `A` list authenticators authenticate `D` list authenticators, so retroactively changing a `D` list authenticators would require retroactive changes to an `A` list authenticators. Changing `A` list authenticators retroactively is not possible because `A` list authenticators are authenticated by the authenticated dictionaries. Therefore changing an `A` list authenticators requires breaking the authenticated dictionaries' membership security. □

D.3 Forward privacy

Lemma 14. *dSEAC is forward private.*

Proof. dSEAC's forward privacy follows from the update leakage $\mathcal{L}_{\mathrm{Upd}}(uid, B)$ for the case that no user is corrupt.[8] In this case, the leakage does not contain any information on keywords contained in B, except for the total number of keywords in B. Hence, dSEAC is forward private. □

[8]The concept of forward privacy does not make any sense if the adversary controls a corrupt user, because then, the adversary can use the corrupt user's knowledge to ask whether a specific keyword is contained in the document collection.