Searchable Encryption with Secure and Efficient Updates

Florian Hahn
SAP, Karlsruhe
Germany
florian.hahn@sap.com

Florian Kerschbaum
SAP, Karlsruhe
Germany
florian.kerschbaum@sap.com

ABSTRACT

Searchable (symmetric) encryption allows encryption while still enabling search for keywords. Its immediate application is cloud storage where a client outsources its files while the (cloud) service provider should search and selectively retrieve those. Searchable encryption is an active area of research and a number of schemes with different efficiency and security characteristics have been proposed in the literature. Any scheme for practical adoption should be efficient – i.e. have sub-linear search time –, dynamic – i.e. allow updates – and semantically secure to the most possible extent. Unfortunately, efficient, dynamic searchable encryption schemes suffer from various drawbacks. Either they deteriorate from semantic security to the security of deterministic encryption under updates, they require to store information on the client and for deleted files and keywords or they have very large index sizes. All of this is a problem, since we can expect the majority of data to be later added or changed. Since these schemes are also less efficient than deterministic encryption, they are currently an unfavorable choice for encryption in the cloud. In this paper we present the first searchable encryption scheme whose updates leak no more information than the access pattern, that still has asymptotically optimal search time, linear, very small and asymptotically optimal index size and can be implemented without storage on the client (except the key). Our construction is based on the novel idea of learning the index for efficient access from the access pattern itself. Furthermore, we implement our system and show that it is highly efficient for cloud storage.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Cryptographic controls*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660297.

Keywords

Searchable Encryption; Dynamic Searchable Encryption; Secure Index; Update

1. INTRODUCTION

Searchable (symmetric) encryption [8, 9, 11, 13, 19, 20, 24, 29] consists of three operations. Encryption transforms a keyword/file pair using a secret key into a ciphertext. Using the secret key one can generate a search token for a specific keyword. Using this token, one can then search in a set of ciphertexts for those that match the keyword. Hence, one can encrypt, but still search without decryption.

The immediate application for searchable encryption is cloud storage where the client outsources its storage, but encrypts its files for confidentiality beforehand and retains the key. The advantage compared to standard encryption is that the cloud can perform the search operation without the key and only return a matching subset for a query. Hence, the client does not have to download the entire data set and search himself. In many cases this is an enormous efficiency gain.

Any practical searchable encryption scheme should be efficient, dynamic and secure. By efficiency we mean sub-linear search time and this is achieved using an (inverted) index. Indices can be difficult to update, particularly if they are encrypted, but efficient dynamic searchable encryption [8, 19, 20, 24] achieves just this. Nevertheless, current efficient, dynamic schemes [20, 24] leak a deterministic function of the keyword during an update, require client storage and additional storage linear in the number of deletes [8] or have an index of the size of the number of documents times the number of keywords [19]. This is a problem, since in long-running systems the majority of data will be later added, changed or deleted. Hence, a long-running system using such a method is either no more secure or very inefficient in storage. Furthermore, since deterministic encryption is very efficient, searchable encryption becomes an unfavorable choice for such cloud storage systems.

In this paper we present the first dynamic searchable encryption scheme with secure and efficient updates. Even under updates, our schemes leaks no more than what can be inferred from the search tokens. Our index is only linear in the number of keywords, hence asymptotically optimal, and client storage (except the key) is optional in our construction[1]. While our time for the first, initial search of a keyword is linear in the number of keywords, we show that

[1]Client storage can be used to speed up additions

it amortizes over multiple searches and is hence practical. We show a theoretic upper bound for amortization in $O(n^2)$ searches in Section 6.3. In our experiments amortization is approached much faster, namely after $0.73n$ searches, which is a much more practical assumption. Furthermore, 84% of all keywords where never searched for and remained semantically secure encrypted.

Hence, our scheme offers a new viable alternative for secure storage in the cloud. It is nearly as efficient as deterministic encryption having the same overall search complexity, but it is clearly more secure leaking only the access pattern of files. We implement our scheme and show that is highly efficient also in absolute performance metrics.

In summary, we contribute a searchable encryption scheme that is

- *dynamic*. Data can be added, deleted and hence changed after the initial outsourcing. In fact, we do not offer a specific operation for initial outsourcing and assume all data is added incrementally.
- *efficient*. Our scheme has asymptotically optimal, sublinear search time. Furthermore, our Java-based implementation shows that a search in a collection with 300.000 keywords and documents can be performed in 70 ms on average. We only require to store 2 cryptographic hash values per keyword and document.
- *secure*. We formalize security using a simulation-based definition. Particularly, we define our own leakage functions which are significantly more restrictive than those of related work. Loosely speaking, our scheme is semantically secure unless a search token has been revealed.

The remainder of the paper is structured as follows. In the next section, we explain the problem of securely updating searchable encryption in more detail and outline our solution approach. In Section 3 we describe the algorithms of our scheme and formally define its security. We present our construction in Section 4, before we prove its security according to our definition in Section 5. We discuss open aspects in Section 6. In Section 7 we present the results of the evaluation of our implementation. Finally, in Section 8 we overview related work, before we summarize our conclusions in Section 9.

2. PROBLEM DESCRIPTION

Searchable encryption consists of three operations: encryption, token generation and search. Encryption takes a plaintext, e.g. a file identifier, a (set of) keyword(s) and a key as input. In this paper we investigate symmetric searchable encryption, but there also exists public key searchable encryption [6]. Encryption produces a ciphertext which can be outsourced to a server, e.g. in the cloud. The key holder can generate a search token for a keyword using the token generation operation. The storage service provider can identify all ciphertexts for a keyword using this search token and the ciphertext(s) in the search operation. He learns which ciphertexts match the query (the access pattern) and multiple ciphertexts can match.

Searchable encryption can secure outsourcing of data by retaining the key at the client. Still due to the search capability efficient retrieval can be implemented. The leakage of the access pattern is key to this efficiency, since it allows to retrieve all ciphertexts in one round. Methods that hide the access pattern come with additional restrictions. Private information retrieval (PIR) [10, 22] can only retrieve one ciphertext. Furthermore computational PIR [22] requires a linear scan of the data [28] and information-theoretic PIR requires multiple servers to store the data [10]. Oblivious RAM [14, 31, 33] accesses also one entry at a time and that even with at least logarithmic overhead. An interesting scheme is proposed by Stefanov et al. that combines ORAM and searchable encryption [30]. This scheme leaks the access pattern, but hides all keywords of updated files. Hence, it has less leakage than our scheme, but it still has logarithmic search time.

Song et al. introduced searchable encryption as a standard semantically secure encryption scheme that only leaks the access pattern of searches [29]. Their scheme still requires a linear scan of all data for searching. Meanwhile, Hacigümüs et al. use deterministic encryption to search efficiently – in sublinear time – in databases [16]. Clearly, deterministic encryption leaks significantly more information than the access pattern and hence is less secure, but more efficient.

Curtmola et al. introduced semantically secure searchable encryption with sublinear search time [11]. Loosely speaking, the basic idea is to construct an index as also used in deterministic encryption. The index is encrypted and contains deterministically encrypted index keywords. Then there is a list of entries where the pointer to the next entry is encrypted with a key specific for the keyword. The search token is a deterministic function of the keyword plus the key for the decryption of the pointers. The storage service provider can look up the deterministic keyword in sublinear time and then decrypt and traverse the list. Still, this is semantically secure unless a search token has been revealed, since each keyword is encrypted deterministically at most once.

Curtmola et al.'s construction builds this index on the client before he uploads the data to the service provider. They introduce an additional operation called *BuildIndex* that takes all plaintexts and keywords as input. Afterwards they allow no more updates until the client builds the next entire index. Kamara et al. propose dynamic searchable encryption where this index can be incrementally updated [20]. They describe an update operation that, loosely speaking, takes the deterministically encrypted keyword, a ciphertext and a token key as input. The service provider can then insert the new ciphertext at the beginning of the list and encrypt the pointer using the token key.

Another contribution starting with Curtmola et al. is the simulation-based security definition. Differently from a game-based security definition as in standard semantic security the simulation-based security uses all leaked information to construct a simulator that produces indistinguishable output. This has the clear advantage that the expected leakage of the encryption is explicitly spelled out. In order to accomodate updates Kamara et al. introduced additional leakage functions: $\mathcal{L}_3$ and $\mathcal{L}_4$ in their paper. Particularly, for additions $\mathcal{L}_3$ leaks a deterministic function of each keyword in a ciphertext. Now, consider a case where the client uploads an empty index and incrementally adds all ciphertexts. In this case, the deterministic keyword function result(s) is (are) revealed for each ciphertext. Hence, in this case dynamic searchable encryption is no more secure than any determin-

Scheme	Search Time	Index Size	Client Storage	Revocation Storage	Update Leakage	Update Cost
[20]	$O(m/n)$	$O(m+n)$	$O(1)$	-	$ID(w)$	$O(m/n)$
[24]	$O(m/n)$	$O(m+n)$	$O(1)$	-	$ID(w)$	$O(m/n)$
[19]	$O(\log \lvert\mathbf{f}\rvert \cdot m/n)$	$O(\lvert\mathbf{f}\rvert \cdot n)$	$O(1)$	-	-	$O(\log \lvert\mathbf{f}\rvert \cdot n)$
[8]	$O(m/n)$	$O(m+n)$	$O(n)$	$O(m)$	-	$O(m/n)$
This paper 1	$O(m/n)$	$O(m+n)$	$O(n)$	-	-	$O(m/n)$
This paper 2	$O(m/n)$	$O(m+n)$	$O(1)$	-	-	$O(m)$

Table 1: Overview over *average complexities* for efficient dynamic searchable encryption schemes. n is the number of unique keywords, m is the total number of keywords, $ID(w)$ is a deterministic identifier of the keyword, $|\mathbf{f}|$ is the number of files

istic encryption, yet has a constant overhead in space and time.

Naveed et al. propose a scheme which trades storage for performance by scattering the stored blocks using hashing instead of encrypting the index [24]. They still leak a deterministic function of the added keywords, i.e. the block where the keyword index is stored. Kamara and Papamanthou fix this problem by using a tree-based construction, but this has index size linear in the number of documents times the number of keywords [19]. Furthermore, constants are quite high, since each index entry (one bit) requires a semantically secure ciphertext. Cash et al. fix this problem by giving each update a counter, but the client has to keep track of these counters [8]. Furthermore, in order to accomodate deletes, they organize those by keeping a revocation list, such that the data may actually never be deleted. Nevertheless, they offer an edit operation that is also indistinguishable from an add operation in case of never before searched keywords. Table 1 gives a comparison of these and our schemes.

The problem we consider in this paper is whether we can update an outsourced storage without leaking anything except the access pattern and with minimal storage overhead. It is important to maintain an index for sublinear search time, since for large data sets linear scans are prohibitive, particularly if they involve a linear number of cryptographic operations. Moreover, it is important to provide storage-efficient updates, since in the long run the majority of the data will have been added after the initial outsourcing. Over the history of computing we have observed an exponential growth in data, such that the initial data set is quickly marginalized.

On a high level our approach works by learning the index from the access pattern. We start with a non-index based searchable encryption scheme that requires linear scans. When we search, we learn the search token which is deterministic and the access pattern. We then start to construct an index using the token and the accessed ciphertexts. When we search the same keyword again we can use the index and search in constant time. Over the long run, the initial linear search time amortizes and we achieve asymptotically optimal search time while leaking nothing except the access pattern.

We show a theoretic analysis with an upper bound for amortization of $O(n^2)$ searches. Yet, in our experiments amortization is approached much faster, namely after $0.73n$ searches.This is an assumption which can be easily met in practice.

Clearly, the access pattern of past searches extends to the future. A search token stays valid and can be used to match against future ciphertexts until the entire system is rekeyed. We have to account for this in our security definition and include the respective leakage. We emphasize that this leakage is notably less than the leakage of the add operation by Kamara et al. [20] or Naveed et al. [24] and differently to their scheme ciphertexts for not previously searched keywords remain semantically secure. Furthermore, this leakage is already part of the most simple construction based on the searchable encryption scheme by Song et al. [29] and it is also not excluded in any of the other efficient, dynamic searchable encryption schemes [8, 19]. In our experiments using real-world search terms, 84% of all keywords were never searched for and hence remained semantically secure encrypted.

We also optionally maintain a history of previous search tokens at the client, such that ciphertexts for previously searched keywords are encrypted differently. We emphasize that this choice is a pure performance optimization and the history could just as well be kept at the service provider at a higher update cost. Furthermore, should the client lose the saved history, he can restore it from the index information of the service provider. We further explain this choice in Section 6.1.

Still, our construction is highly efficient and provides practical performance. We implement our system and a search in collection with 300.000 keywords and documents can be performed in 70 ms on average. We require very little storage overhead and only need to store 2 cryptographic hash values per keyword and document. We provide the detailed performance results in Section 7.

3. DEFINITIONS

The set of binary strings of length n is denoted as $\{0,1\}^n$, the set of all finite binary strings is denoted as $\{0,1\}^*$. Given a binary string u, we denote $\texttt{len}(u)$ as its bit length. Given two binary strings u, v, the concatenation is written as $u||v$. The notation $[1,n]$ with $n \in \mathbb{N}$ denotes the integer set $\{1,\ldots,n\}$. We denote the output z of a (possibly probabilistic) algorithm $\mathcal{A}$ as $z \leftarrow \mathcal{A}$. Sampling uniformly random from a set X is denoted as $x \leftarrow X$.

Throughout, λ will denote the security parameter. A function $f : \mathbb{N} \to \mathbb{R}$ is negligible in x if for every positive polynomial $p(\cdot)$ there exists a x_0 such that for all $x > x_0$, $f(x) < 1/p(\cdot)$.

We assume each file f having a unique file identifier $ID(f)$, each file consists of words that is $f = (w_1, \ldots, w_{\texttt{len}(f)})$ with $w_i \in \{0,1\}^*$. For a fileset $\mathbf{f}$ we denote $\texttt{len}(\mathbf{f})$ as the number of files in $\mathbf{f}$. Given a keyword w we write $\mathbf{f}_w$ as the subset of all files $\mathbf{f}$ that contain w. In addition, the set of all file identifiers that contain this keyword w is denoted by $\mathbf{I}_w$ more formally it is defined as $\mathbf{I}_w = \{ID(f_i) : f_i \in \mathbf{f}_w\}$.

As mentioned in Section 1, our scheme does not offer an operation for initial outsourcing a set of files but starts with an empty search index γ. The service provider's search index γ and collection of encrypted files $\mathbf{c}$ are updated by file specific add tokens α_f for file f and its encryption.

To perform a search query for keyword w, the client generates an deterministic search token τ_w that is handed to the service provider. For simplicity of the exposition we assume that all generated search tokens are given to the service provider, i.e. if a search token has been created by the client the service provider gains knowledge of it.

Finally, to delete a file f the client simply passes its file identifier $ID(f)$ to the service provider.

DEFINITION 1 (SUISE). *A securely updating index-based searchable encryption scheme is a tuple of eight (possibly probabilistic) polynomial-time algorithms* SUISE = (Gen, Enc, SearchToken, Search, AddToken, Add, Delete, Dec) *such that:*

> $(K, \gamma, \sigma) \leftarrow \text{Gen}(1^\lambda)$*: is a probabilistic algorithm that takes as input a security parameter λ and outputs a secret key K, a (still empty) search index γ and a (still empty) search history σ.*
>
> $c \leftarrow \text{Enc}(K, f)$*: is a probabilistic algorithm that takes as input a secret key K and a file f. It outputs an encrypted file c.*
>
> $(\sigma', \tau_w) \leftarrow \text{SearchToken}(K, w, \sigma)$*: is a (possibly probabilistic) algorithm that takes as input a secret key K, a keyword w and search history σ. It outputs a new search history σ' and a search token τ_w.*
>
> $(\boldsymbol{I}_w, \gamma') \leftarrow \text{Search}(\tau_w, \gamma)$*: is a deterministic algorithm that takes as input a search token τ_w, a sequence of encrypted files $\mathbf{c}$ and a search index γ. It outputs a sequence of identifiers $\boldsymbol{I}_w$ and an updated search index γ'.*
>
> $\alpha_f \leftarrow \text{AddToken}(K, f, \sigma)$*: is a (possibly probabilistic) algorithm that takes as input a secret key K, a file f and a search history σ. It outputs an add token α_f.*
>
> $(\boldsymbol{c}', \gamma') \leftarrow \text{Add}(\alpha_f, c, \boldsymbol{c}, \gamma)$*: is a deterministic algorithm that takes as input an add token α_f, an encrypted file c, a sequence of encrypted files $\mathbf{c}$ and a search index γ . It outputs an updated search index γ' and an updated sequence of encrypted files $\boldsymbol{c}'$.*
>
> $(\boldsymbol{c}', \gamma') \leftarrow \text{Delete}(ID(f), \boldsymbol{c}, \gamma)$*: is a deterministic algorithm that takes as input an identifier $ID(f)$ of the file that shall be removed, a sequence of encrypted files $\mathbf{c}$ and a search index γ. It outputs an updated sequence of encrypted files $\boldsymbol{c}'$ and an updated search index γ'.*
>
> $f \leftarrow \text{Dec}(K, c)$*: is a deterministic algorithm that takes as input an encrypted file c and a key K. It outputs the decrypted file f.*

In Figure 1 we show the protocols between client and server combining these algorithms into interaction patterns.

A dynamic searchable encryption scheme is called *correct* if for all $\lambda \in \mathbb{N}$, all keys K generated by $\text{Gen}(1^\lambda)$, and all sequences of add, delete and search operations on search index γ, every search operation returns the correct set of files (except with negligible probability).

In an ideal scenario, searchable encryption is implemented in a way where the service provider learns absolutely nothing about either the files or the search queries. As mentioned in Section 2, there are methods to achieve this strict security goals, but these come along with huge overhead. By allowing the sever to learn particular information (e.g. the access pattern) we can construct more efficient searchable encryption schemes. To address this small knowledge the service provider gains, we follow the approach of [8, 11, 19, 20, 24] and use leakage functions. The additional knowledge the provider gains by getting ciphertexts and (add or search) tokens is defined by these functions.

As noticed in [11], there is a difference between security against *adaptive* chosen keyword attacks (CKA2) and *non-adaptive* chosen keyword attacks (CKA 1), that must be taken into account for security analyses. Security against CKA2 guarantees security even when the client's generated query depend on results of previous queries and the search index. In contrast, security against CKA1 guarantees security only when all queries generated by the client are independent of previous queries and the search index. Our construction achieves the stronger notion of CKA2 security, that is modified in a way suggested by Kamara et al. in [20] to fit into the scenario of dynamic SSE.

DEFINITION 2. *Let* SUISE = (Gen, Enc, SearchToken, Search, AddToken, Add, Delete, Dec) *be a securely updating index-based searchable encryption scheme. Consider the following experiments with stateful attacker $\mathcal{A}$, stateful simulator $\mathcal{S}$ and stateful leakage functions $\mathcal{L}_{search}, \mathcal{L}_{add}, \mathcal{L}_{encrypt}$.*

> $\mathbf{Real}^{SSE}_{\mathcal{A}}(\lambda)$: *the challenger runs $\text{Gen}(1^\lambda)$ to get the tuple (K, γ, σ). The adversary $\mathcal{A}$ makes a polynomial number of adaptive queries $q \in \{w, f_1, f_2\}$ and for each query q the challenger generates either a search token $\tau_w \leftarrow \text{SearchToken}(K, w, \sigma)$, an add token $\alpha_f \leftarrow \text{AddToken}(K, f_1, \sigma)$, or a file encryption $c \leftarrow \text{Enc}(K, f_2)$. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.*
>
> $\mathbf{Ideal}^{SSE}_{\mathcal{A},\mathcal{S}}(\lambda)$: *the simulator sets up its internal environment. The adversary $\mathcal{A}$ makes a polynomial number of adaptive queries $q \in \{w, f_1, f_2\}$ and for each query q the simulator is given the appropriate leakage i.e. either given $\mathcal{L}_{search}(\boldsymbol{f}, w)$, $\mathcal{L}_{add}(\boldsymbol{f}, f_1)$ or $\mathcal{L}_{encrypt}(f_2)$. $\mathcal{S}$ returns the appropriate token $\widetilde{\tau_w}$, $\widetilde{\alpha_f}$ or a ciphertext $\widetilde{c}$. Finally, $\mathcal{A}$ returns a bit b that is output by the experiment.*

We say SUISE *is $(\mathcal{L}_{search}, \mathcal{L}_{add}, \mathcal{L}_{encrypt})$-secure against adaptive dynamic chosen-keyword attacks if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a probabilistic polynomial-time simulator $\mathcal{S}$ so that advantage of $\mathcal{A}$ defined as*

$$\left|\Pr\left[\mathbf{Real}^{SSE}_{\mathcal{A}}(\lambda) = 1\right] - \Pr\left[\mathbf{Ideal}^{SSE}_{\mathcal{A},\mathcal{S}}(\lambda) = 1\right]\right|$$

is negligible in λ.

4. IMPLEMENTATION

On a high level our construction works by learning the index from the access pattern. Initially we maintain a regular index, i.e. for each document we store its (encrypted) keywords. We denote this index γ_f. Once a keyword is searched, we move all file identifiers to an inverted index for

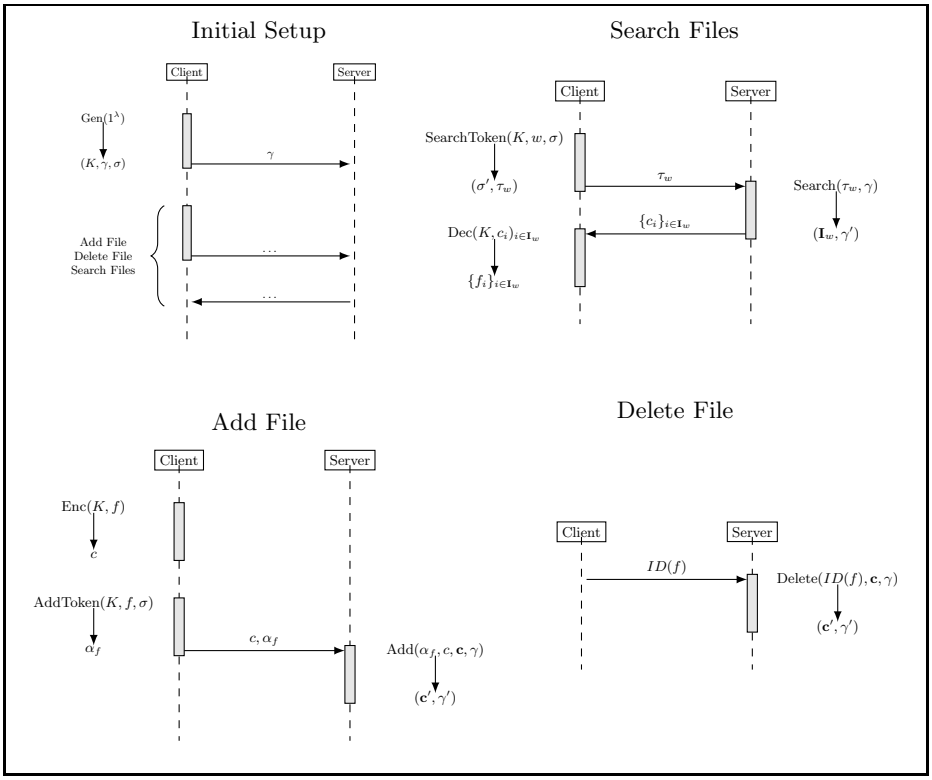

Figure 1: Use of algorithms from Definition 1 for realizing the complete protocol.

the keyword, i.e. the keyword is now the key to the index. The keyword is also encrypted; it is the search token. We denote this index γ_w.

Clearly, we have to accomodate future updates for keywords that have already been searched. These keywords – and their corresponding file identifiers – have been moved to the inverted index. Hence, an update needs to update the inverted index, or we need to always search both indices, completely ruining search time. In this section we present the option to maintain the search history at the client. The client checks whether a keyword has been searched and tells the server to include it in the inverted index. This is a pure performance optimization. We present the alternative option to maintain the search history on the server at a higher update cost in Section 6.1.

For our implementation we use several data structures including lists and (chained) hash tables. For list $\mathbf{l}$ we denote $\texttt{len}(\mathbf{l})$ for the number of elements in $\mathbf{l}$ and $x \in \mathbf{l}$ iff value x is stored in list $\mathbf{l}$. Accessing the element at position i is written as $\mathbf{l}[i]$. A hash table T stores values v associated with keys k, written as $T[k] = v$. We write $v \in T$ if there is a key k so that $T[k] = v$. For our implementation it is crucial that it is feasible to access a value v with corresponding key k stored in a hash table in constant time. If the values stored in the hash table are lists, we call it a chained hash table.

Given an IND-CPA secure secret-key encryption scheme $\text{SKE} = (\mathcal{G}^{\text{IND-CPA}}, \mathcal{E}^{\text{IND-CPA}}, \mathcal{D}^{\text{IND-CPA}})$, a pseudorandom number generator G that outputs random numbers with bit length λ and a pseudorandom functions $F : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$, and a random oracle $H : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ we construct dynamic SSE scheme SUISE = (Gen, Enc, SearchToken, Search, AddToken, Add, Delete, Dec) as follows:

- $(K, \gamma, \sigma) \leftarrow \text{Gen}(1^\lambda)$: sample two λ-bit strings $k_1 \leftarrow \{0,1\}^\lambda$ and $k_2 \leftarrow \mathcal{G}^{\text{IND-CPA}}(1^\lambda)$. In addition, create two empty chained hash tables γ_f, γ_w and an empty set σ. Output (K, γ, σ), with $K = (k_1, k_2)$ and $\gamma = (\gamma_f, \gamma_w)$.
- $c \leftarrow \text{Enc}(K, f)$: parse key $K = (k_1, k_2)$ and output $c = \mathcal{E}^{\text{IND-CPA}}_{k_2}(f)$.
- $(\tau_w, \sigma') \leftarrow \text{SearchToken}(K, w, \sigma)$: parse key $K = (k_1, k_2)$ calculate $F_{k_1}(w) = \tau_w$ set $\sigma' = \sigma \cup \{\tau_w\}$ and output (τ_w, σ').
- $(\mathbf{I}_w, \gamma') \leftarrow \text{Search}(\tau_w, \gamma)$: parse search index $\gamma = (\gamma_w, \gamma_f)$ and check if there is an entry for τ_w in γ_w.
  - If yes, then set $\mathbf{I}_w = \gamma_w[\tau_w]$ and $\gamma'_w = \gamma_w$.
  - Otherwise create an empty list $\mathbf{I}_w$ and do for every $\overline{c} \in \gamma_f$:
    1. for every $c_i \in \overline{c}$ that is $i \in [1, \texttt{len}(c)]$, set $c_i = l_i || r_i$ and check if $H_{\tau_w}(r_i) = l_i$. If yes then insert $ID(f)$ that corresponds to $\overline{c}$ into $\mathbf{I}_w$.

    Update γ'_w by creating an entry $\gamma_w[\tau_w] = \mathbf{I}_w$

  Output $\mathbf{I}_w$ and (an updated version of) $\gamma' = (\gamma'_w, \gamma_f)$.
- $\alpha_f \leftarrow \text{AddToken}(K, f, \sigma)$: parse $K = (k_1, k_2)$. For file f that consists of a sequence of words create a list $\overline{f}$ of *unique* words $f \supseteq \overline{f} = (w_1, \ldots, w_{\texttt{len}(\overline{f})})$. Generate

a sequence of pseudorandom values $s_1, \dots s_{\mathtt{len}(\overline{f})}$ with PRNG G and create an empty list $\mathbf{x}$. For every word $w_i \in \overline{f}$ do the following:

1. compute the corresponding search token $\tau_{w_i} = F_{k_1}(w_i)$
2. if this search token was used for a previous search: if $\tau_{w_i} \in \sigma$, add τ_{w_i} to $\mathbf{x}$.
3. set $c_i = H_{\tau_{w_i}}(s_i)||s_i$

Now sort $\overline{c} = (c_1, \dots, c_{\mathtt{len}(\overline{f})})$ in lexicographic order and set $\alpha_f = (ID(f), \overline{c}, \mathbf{x})$. Output α_f.

- $(\mathbf{c}', \gamma') \leftarrow \text{Add}(\alpha_f, c, \mathbf{c}, \gamma)$: parse $\alpha_f = (ID(f), \overline{c}, \mathbf{x})$, $\gamma = (\gamma_w, \gamma_f)$ and set $\gamma_f[ID(f)] = \overline{c}$. In addition, for every $x_i \in \mathbf{x}$ add $ID(f)$ to $\gamma_w[x_i]$. Update the ciphertexts $\mathbf{c}$ to $\mathbf{c}'$ by adding c. Output $\mathbf{c}'$ and the updated version $\gamma' = (\gamma_w, \gamma_f)$.
- $(\mathbf{c}', \gamma') \leftarrow \text{Delete}(ID(f), \mathbf{c}, \gamma)$: parse $\gamma = (\gamma_w, \gamma_f)$, check for every list $\mathbf{e}$ saved in γ_w if $ID(f) \in \mathbf{e}$ and remove $ID(f)$ in this case from $\mathbf{e}$. Remove ciphertext c corresponding to $ID(f)$ from $\mathbf{c}$ and remove $\gamma_f[ID(f)]$ from γ_f. Output an updated set of encrypted files $\mathbf{c}$ and an updated search index $\gamma' = (\gamma_w, \gamma_f)$.
- $f \leftarrow \text{Dec}(K, c)$: parse $K = (k_1, k_2)$ and output $f = \mathcal{D}^{\text{IND-CPA}}_{k_2}(c)$.

Our scheme provides correctness with all but negligible probability. When we have a collision of H, Search outputs a false index $\mathbf{I}_w$ that is for two different search tokens τ, τ' and two different random numbers s, s' we have $H(\tau, s) = H(\tau', s')$. Since H is a random oracle with image size $\{0,1\}^\lambda$, a collision occurs for N queries with probability proportional to $N^2 2^{-\lambda}$ and therefore is negligible for N polynomial in λ. Furthermore, these collisions result in false positive answers for $\mathbf{I}_w$ and can be filtered out by the client repeating the search on all decrypted files with IDs contained in $\mathbf{I}_w$. Exactly the same argument holds for a collision of the pseudorandom function F, that is generating two equivalent search tokens for different words.

5. SECURITY

As mentioned before, some operations leak particular information to the service provider. In detail, we use three leakage functions $\mathcal{L}_{\text{search}}, \mathcal{L}_{\text{add}}, \mathcal{L}_{\text{encrypt}}$ defined as follows:

$$\begin{aligned} \mathcal{L}_{\text{search}}(\mathbf{f}, w) &= (\text{ACCP}_t(w), ID(w)) \\ \mathcal{L}_{\text{add}}(\mathbf{f}, f) &= (ID(f), \mathtt{len}\left(\overline{f}\right), \text{SRCH\_HIS}_t(\overline{f})) \\ \mathcal{L}_{\text{encrypt}}(f) &= \mathtt{len}\left(f\right) \end{aligned}$$

where $\text{ACCP}_t(w)$ is the access pattern at time t defined as set $\{ID(f_i) : w \in f_i \text{ and } f_i \in \mathbf{f}\}$, $\overline{f}$ is the set of unique words in file f, and $\text{SRCH\_HIS}_t(\overline{f})$ is the set of IDs of all searched words until time t that also appear in $\overline{f}$.

Since search tokens τ are deterministic, an attacker is able to link generated search tokens with words, although she does not know what the plain word is. This is denoted by $ID(w)$ in our leakage functions.

Now we are ready to proof the following theorem:

THEOREM 1. *If the used secret key encryption scheme* SKE *is IND-CPA secure, F is a pseudorandom function and G is a pseudorandom generator, then* SUISE *as described in Section 4 is $(\mathcal{L}_{search}, \mathcal{L}_{add}, \mathcal{L}_{encrypt})$-secure against adaptive dynamic chosen-keyword attacks in the random oracle model.*

PROOF. We describe a polynomial time simulator $\mathcal{S}$ for which the advantage of any PPT attacker $\mathcal{A}$ to distinguish between the output of $\mathbf{Real}^{\text{SSE}}_{\mathcal{A}}(\lambda)$ and $\mathbf{Ideal}^{\text{SSE}}_{\mathcal{A},\mathcal{S}}(\lambda)$ is negligible. Our simulator adaptively simulates a search index $\widetilde{\gamma}$ with the additional information given by the leakage functions.

1. Setting up the environment:

   $\mathcal{S}$ creates an empty list $\widetilde{\sigma}$ as simulated search history, an empty simulated search index $\widetilde{\gamma} = (\widetilde{\gamma_w}, \widetilde{\gamma_f})$ consisting of two empty hash tables, and an empty dictionary ρ to keep track of queries to the random oracle. A key $k_3 \leftarrow \mathcal{G}^{\text{IND-CPA}}(1^\lambda)$ is sampled for simulating encryption of files. A chained hash table C is used to keep track of tuples $(j, \widetilde{\tau})$ consisting of previously simulated search indexes and simulated search tokens for each individual file. In detail, an entry of C consists of a linked list, we denote C_{f_i} as the linked list for file f_i, that is stored at hash table entry $C[ID(f_i)]$. Furthermore, other empty hash tables T and A are created, where T is needed to keep track of the assignment of simulated search tokens $\widetilde{\tau_w}$ for word w with $ID(w)$ and A is needed to keep track of simulated add tokens $\widetilde{\alpha_f}$ for already added files with $ID(f)$.

2. Simulating search tokens $\widetilde{\tau}$ with given leakage

   $$\mathcal{L}_{\text{search}}(\mathbf{f}, w) = (\text{ACCP}_t(w), ID(w)),$$

   the simulator checks if $ID(w)$ is in T i.e. if a search token for this word was queried before.

   - If this is the case, $\mathcal{S}$ outputs $T[ID(w)]$.
   - Otherwise, a random bit string $\widetilde{\tau} \leftarrow \{0,1\}^\lambda$ is chosen and stored at $T[ID(w)]$ and added to $\widetilde{\sigma}$. For every $ID(f_i) \in \text{ACCP}_t(w)$ the simulator sets J_{f_i} as the set of all first components of C_{f_i} more formally set $J_{f_i} = \{j_l : (j_l, \tau_l) \in C_{f_i} \text{ with } 0 \leq l \leq |C_{f_i}|\}$. Then the simulator chooses a random index $j_i \leftarrow [1, |\widetilde{\gamma_f}[ID(f_i)]|] \setminus J_{f_i}$ and adds the tuple $(j_i, \widetilde{\tau})$ to the list C_{f_i}. Finally, $\mathcal{S}$ outputs $\widetilde{\tau}$.

3. Simulating add $\widetilde{\alpha}$ tokens with given leakage

   $$\mathcal{L}_{\text{add}}(\mathbf{f}, f) = (ID(f), \mathtt{len}\left(\overline{f}\right), \text{SRCH\_HIS}_t(\overline{f})),$$

   $\mathcal{S}$ checks if there is an entry at $A[ID(f)]$ i.e. if an add token for file f with $ID(f)$ was simulated before.

   - If an add token $\widetilde{\alpha_f}$ for file f with $ID(f)$ was requested before, the simulator outputs $A[ID(f)]$.
   - If, on the other hand, this file was not added before, the simulator chooses for every $i \in [1, \mathtt{len}\left(\overline{f}\right)]$ a random bit string $\widetilde{s_i} \leftarrow \{0,1\}^{2\lambda}$, and sorts this generated set $(\widetilde{s_1}, \dots, \widetilde{s}_{\mathtt{len}(\overline{f})})$ in lexicographic order to get $\widetilde{s}$ and stores this at $\widetilde{\gamma_f}[ID(f)]$. In addition, an empty list $\widetilde{\mathbf{x}}$ is created and for every $ID(w) \in \text{SRCH\_HIS}_t(\overline{f})$ the token $\widetilde{\tau_w} = T[ID(w)]$ is looked up, added to $\widetilde{\mathbf{x}}$ and $ID(f)$ is added to $\widetilde{\gamma_w}[\widetilde{\tau_w}]$. $\mathcal{S}$ creates a temporary set J and for all $l \in [1, \mathtt{len}\left(\widetilde{\mathbf{x}}\right)]$ a random fake index $j_l \leftarrow$

$[1, |\widetilde{\gamma_f}[ID(f_i)]|] \setminus J$ is sampled and added to J. This index j_l is marked as full in the list of used search indexes by adding the tuple $(j_l, \widetilde{\tau}_l)$ to C_f in the chained hash table C, where $\widetilde{\tau}_l = \widetilde{\mathbf{x}}[l]$. Finally, $\mathcal{S}$ outputs $\widetilde{\alpha_w} = (ID(f), \widetilde{s}, \widetilde{\mathbf{x}})$ and stores this simulated token at $A[ID(f)]$.

4. Simulating encryption with given leakage

$$\mathcal{L}_{\text{encrypt}}(f) = \texttt{len}(f),$$

the simulator outputs $\widetilde{c} \leftarrow \mathcal{E}_{K_3}^{\text{IND-CPA}}(0^{\texttt{len}(f)})$.

5. Answering random oracle queries: given query (k, r), the simulator checks if this query was submitted before i.e. if there is an entry $l = \rho[k||r]$.

   - If this is the case, set $l = \rho[k||r]$.
   - Otherwise, $\mathcal{S}$ checks if key k is linked with some $ID(w)$ i.e. if there is an entry $k = T[ID(w)]$ for some $ID(w)$. If there is no used search token, a random bit string $l \leftarrow \{0,1\}^\lambda$ is sampled and $\rho[k||r] = l$ is set to stay consistent for future queries.

     If, on the other hand, k is linked with some $ID(w)$, for every $ID(f_i) \in \widetilde{\gamma_w}[k]$ the simulator looks up the tuple $(j, k') \in C_{f_i}$ where $k' = k$. Then the value $\widetilde{s_j} \in \{0,1\}^{2\lambda}$ is set as the j-th entry of $\widetilde{s} = \widetilde{\gamma_f}[ID(f_i)]$ and divided in two λ-bit strings $l'||r' = \widetilde{s_j}$.
     - $\mathcal{S}$ checks if $r' = r$, sets $l = l'$ in this case and stores l at $\rho[k||r]$.
     - If there was no fitting r' for any $ID(f_i) \in \widetilde{\gamma_w}[k]$, a random $l \leftarrow \{0,1\}^\lambda$ is sampled and stored at $\rho[k||r]$ to stay consistent for future queries.

   Finally, l is returned.

The indistinguishability of a simulated search token $\widetilde{\tau}$ and a real search token τ follows from the pseudorandomness of F. Also, the indistinguishability of a simulated search history $\widetilde{\sigma}$ and a real search history σ follows from the pseudorandomness of F. The indistinguishability of a simulated add token $\widetilde{\alpha}$, especially of $\widetilde{s}, \widetilde{\mathbf{x}}$, and a real add token α follows from the pseudorandomness of G and F. The indistinguishability of a simulated ciphertext $\widetilde{c}$ and real ciphertext c follows from the IND-CPA security of our used secret key encryption. Since we choose the output of our simulated random oracle H either totally random or out of a predefined domain, that itself is generated in a random way, our random oracle is indistinguishable from a pseudo-random function. □

6. DISCUSSION

6.1 Why maintain a search history at the client?

We maintain a history of previously used search tokens at the client and use it during the add operation. The client creates the corresponding search tokens immediately as the deterministic identifier of the keyword. Hence, the service provider can include it in the index. Note that the search token is not necessarily part of the add token, rather it could be randomized. In order to check whether a randomized add token corresponds to a search token, the service provider would need to check all previous search tokens. Only then, it could convert the randomized add token to the deterministic search token. To the contrary, the client can compute both – search and add token of the inserted keyword – and simply look up the search token in the history. Hence, the cost of one insertion is $O(len(\overline{f}))$ using a history at the client and $O(len(\overline{f})|SRCH\_HIS_t(\mathbf{f})|)$ using a history at the service provider. Our solution using no client storage (except the key) modifies the Add and AddToken operations as follows:

- $\alpha_f \leftarrow \text{AddToken}(K, f, \sigma)$: parse $K = (k_1, k_2)$. For file f that consists of a sequence of words create a list $\overline{f}$ of *unique* words $f \supseteq \overline{f} = (w_1, \ldots, w_{\texttt{len}(\overline{f})})$. Generate a sequence of pseudorandom values $s_1, \ldots s_{\texttt{len}(\overline{f})}$ with PRNG G. For every word $w_i \in \overline{f}$ set $c_i = H_{\tau_{w_i}}(s_i)||s_i$ with $\tau_{w_i} = F_{k_1}(w_i)$. Now sort $\overline{c} = (c_1, \ldots, c_{\texttt{len}(\overline{f})})$ in lexicographic order and set $\alpha_f = (ID(f), \overline{c})$. Output α_f.
- $(\mathbf{c}', \gamma') \leftarrow \text{Add}(\alpha_f, c, \mathbf{c}, \gamma)$: parse $\alpha_f = (ID(f), \overline{c})$, $\gamma = (\gamma_w, \gamma_f)$ and set $\gamma_f[ID(f)] = \overline{c}$. In addition, for each $\tau_{w_i} \in \gamma_w$ and each $c_j \in \overline{c}$ set $c_j = l_j||r_j$ and check if $H_{\tau_{w_i}}(r_j) = l_j$. If yes, add $ID(f)$ to $\gamma_w[w_i]$. Update the ciphertexts $\mathbf{c}$ to $\mathbf{c}'$ by adding c. Output $\mathbf{c}'$ and the updated version $\gamma' = (\gamma_w, \gamma_f)$.

The history at the client is the same as the index words $ID(w)$ in inverted index γ_w at the service provider. Hence the client can always restore its history by downloading these from the server. Moreover, let the number of unique keywords be n, then the size of the history will never exceed $O(n)$ independent of the number of searches performed.

6.2 How to hide the number of unique keywords per file?

Our leakage definition for the add operation still includes the identifier of the file ($ID(f)$) and the number of unique keywords in that file ($len(\overline{f})$). One can hide both by adding a level of indirection. First encrypt each file f, resulting in the identifier $ID(f)$. Then for each $w_i \in \overline{f}$ encrypt file $f' = \{ID(f)\}$ resulting in unique identifier $ID(f')$. One can now create the add token for $ID(f')$ and w_i. A simulator for the add token operation is simple to derive from our simulator: $ID(f')$ – which is unique in the system – replaces $ID(f)$ and is leaked instead, but $len(\overline{f'}) = 1$ and can hence be omitted.

6.3 Search Time Analysis

In our algorithm the first search for a keyword requires a linear scan, but subsequent searches are (almost) constant. Hence, the initial overhead amortizes and we reach asymptotically optimal search time for long-running systems. In this section we analyse the required number of searches until we reach this optimum.

Let n be the number of unique keywords stored in the ciphertexts; let m be the total number of stored keywords in the ciphertext. Once we have created an index entry for a keyword, our search complexity is m/n: We have a constant lookup in the hash table and return m/n ciphertexts on average. An initial search can take up to m search (lookup) operations and there can be at most n of those. Hence, the initial effort is (upper) bounded by mn. We are interested in

the number N of searches such that the amortized cost becomes optimal. Since we need to return at least m/n entries, this is the lower bound optimum. The cost is asymptotically optimal, if there exists a constant c, such that the cost is at most the optimum times c. The amortized cost is the cost for inital searches (mn) divided by the number of searches plus the cost for one subsequent search:

$$\frac{mn}{N} + \frac{m}{n} \leq c\frac{m}{n}$$

We conclude from this formula that we need at least $N \geq n^2$ searches until our cost is asymptotically optimal. The constant $c = 2$ is low and we need at most 0.5 cryptographic hash operation on the server on average. Read (search) requests dominate many systems like databases, such that this number can be quickly reached in practice.

7. PERFORMANCE RESULTS

The following experiments were implemented in Java 7. Either operations performed by the server, or operations performed by the client, were executed on an Intel Xeon 1230v3 CPU 3.30GHz with 8GB RAM running Windows 8. To minimize I/O access time all files used in our simulations are loaded into main memory before starting measurements.

For the implementation of our keyed random function F and our random oracle H we use the implementation of HMAC-SHA-1 that is contained in the default Java library. Also contained in the default Java library, we use SHA1PRNG as random number generator G that outputs a PRN with length of 160 bits.

7.1 On the client's side

One main argument for outsourcing data is the use of slow and weak hardware on client's side. To show our SUISE scheme feasible for this scenario, we simulated the creation of add tokens and search tokens for $250,000$ random words per run. We repeated each creation run 100 times and present the mean value of these 100 runs for creating one token. We simulated both versions for storing the search history that is storing it on the client or storing it on the server.

The cost for creating search tokens depends on the cost of generating one HMAC-SHA-1 mainly. Creating an add token without checking the search history (so let the server check for indexed words used in previous search queries) needs two HMAC operations and one random number. If the client has to check the search history, the cost for add token generation also depends on the size of search history. For our simulation we filled the search history with 0, $100,000$ and $1,000,000$ unique words. By using Java's HashSet as search history, the lookup time can be minimized. Remember that the search history contains *unique* search words used before and stays quite small (see Figure 3) in relation to the number of search queries.

We omitted time measurements for encryption and decryption operations, since encrypting and decrypting files with secret key encryption in an IND-CPA secure way is a well studied problem. Furthermore, special clients may have highly specialized hardware for performing a special kind of secret key encryption scheme.

7.2 On the server's side

All our simulations ran single threaded, but can easily be executed in parallel. By dividing the search index γ_f into subsets and search these subsets on different cores it is possible to speed up searches for search tokens that were not searched before. In our test scenario all operations ran on one machine so we were able to ignore latency through network transfers that may occur in practice application.

operation	$\lvert\sigma\rvert$	time [μs]
SearchToken	-	1.14
AddToken	0	4.77
AddToken	10^5	5.06
AddToken	10^6	5.47

Figure 2: Average duration for creating one token for one word.

We omit benchmarks for Add and Delete since the runtime of these tasks depend on the chosen methods for creating indexes and updating these indexes. In addition, one can interpret these operations as storing, accessing, adding and deleting plaintext in an efficient way, because no cryptographic primitives are used there. Either cryptographic primitives are used on the client before and benchmarked there (e.g. creating add tokens) or are not needed at all.

So, the runtime of Search is discussed in the following. For our experiments we added 50 ebooks downloaded from project Gutenberg [1]. Before adding these files, all words were transformed to lower case and punctuation was removed. Our complete fileset $\mathbf{f}$ consisted of $3,654,417$ words separated by whitespaces after this transformation. Removing words that appear multiple times in one file resulted in a filest containing $337,724$ words so our index γ_f had size $337,724$. Altogether $95,465$ words were indexed i.e. $95,465$ different search tokens would result in at least one positive match. To simulate realistic search queries we use a list of word frequencies from [2] which represent real world search queries but omitted the first 100 entries that mainly contained pronouns and prepositions. This word frequency list contains about $400,000$ words and our search words are chosen in a random fashion weighted according to their frequency.

In order to benchmark search operations Search at the service provider we generate 5000 random search tokens using the probability distribution explained above. The mean search time for these 5000 search tokens results in one measurement point. A complete benchmark run consists of $75,000$ search queries, i.e. 15 values are measured per run. In total, we repeated these benchmark runs 10 times and plotted the average and the error bars provide the standard derivation of these 10 runs. The average time for an initial, linear search was 414.38 ms. The average time for a second, constant time search was 0.01 ms.

Figure 3 shows the size of the search history over the time of the experiment. It also depicts the decreasing number of newly generated search tokens that were not contained in the search history before. Denoted by the white part, every bar represents the size of search history σ before the 5000 search queries. The grey part represents the amount of queried search tokens that were not known before these 5000 search queries. So, combining the white and the grey part shows the size of search history σ after executing these 5000 queries. Figure 3 demonstrates the decreasing amount of newly generated search tokens with increasing amount of

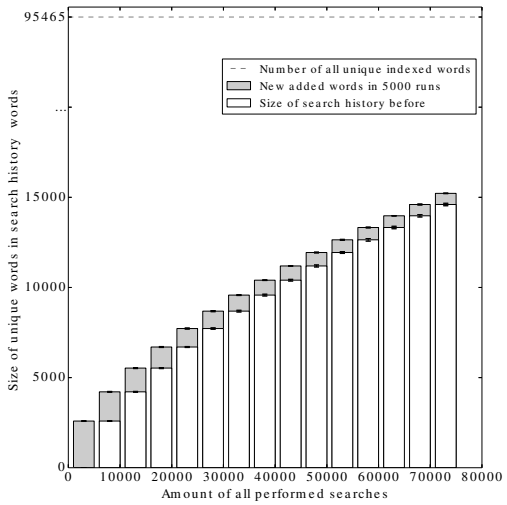

Figure 3: Unique search tokens queried; grey represents new search tokens not asked before.

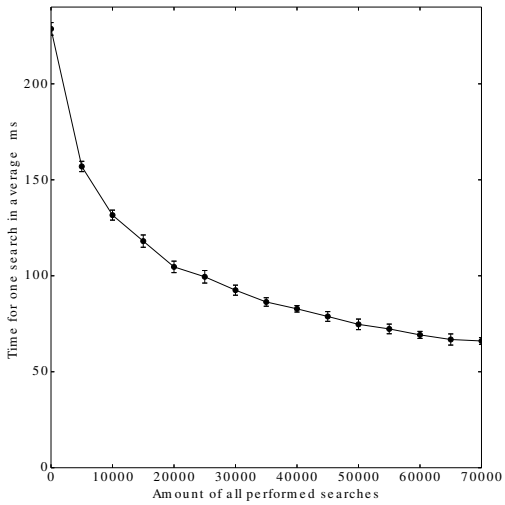

Figure 4: Mean query time for one random generated search token of 5000.

total search queries. On the one hand, due to this effect the majority of our randomized add tokens remain randomized. At the end of the experiment less than 16% of the unique keywords are in the inverted index γ_w and hence encrypted deterministically. On the other hand, this results in decreasing search time, because already searched tokens can be looked up in this inverted index.

This effect is presented in Figure 4 in more detail. It shows the mean search time in ms for blocks of 5000 searches over the time of the experiment. One bar at position i in Figure 3 can be linked with the point in Figure 4 at position i on x-axes. At the beginning the service provider did not know any search tokens so that the search history σ is empty. Given a search token in this situation, the service provider needed to check the index of every file i.e. every list representing one file in γ_f had to be checked value by value. With an increasing size of the search history an increasing number of search tokens were indexed in our reverse index γ_w and hence the service provider was able to answer these queries much faster. Our results also show that the optimal search time is reached much faster than in n^2 searches, since we only perform 70.000 searches for more than 95.000 unique keywords.

8. RELATED WORK

Song et al. introduced searchable encryption [29]. It provides a more secure alternative for searching compared to deterministic encryption. Deterministic encryption is useful for searching in outsourced databases, since it does not require to modify the database engine for queries or updates [16, 17]. Recent results, such as CryptDB [25], show that its performance overhead is negligible and only a minority of database columns need to be encrypted in deterministic encryption. Motivated by these use cases Bellare et al. investigated deterministic encryption in a public key setting [5]. Still, they conclude that searchable encryption, as also proposed in this paper, could help securing the searched data.

A number of works investigated efficiency increases for searchable encryption. Goh first proposed the use of indexes [13]. Deterministic encryption allows the construction of (most) indexes on the ciphertext – without modification of the algorithm –, but searchable encryption requires an adjustment of the indexing method. Curtmola et al. were the first to use inverted indexes [11]. While constructing the index can be performed on the plaintext, modifying it without leakage is difficult. We discussed in detail in Section 2 the different drawbacks of the dynamic searchable encryption schemes [8, 19, 20, 24]. We did not consider the scheme in [32], since it does not contain a formal security proof. The scheme proposed in this paper is the first with asymptotically optimal search time, asymptotically optimal storage cost and no leakage on updates.

Searchable encryption has also been extended in a number of ways. Boneh et al. introduced public key searchable encryption based on identity-based encryption [6]. Another direction is complex queries for conjunctive [15] and disjunctive [7] keyword combinations. Recent results achieved significant efficiency gains for these complex queries [9, 21]. Extending our method to complex queries is subject of future work.

Searchable encryption has also been combined with proxy re-encryption [12, 26]. Popa et al. also demonstrate another interesting application of re-encryptable, searchable encryption for securing web applications [26]. Commercial offerings in this space currently focus on deterministic encryption [3, 4], since they do not require to rewrite the web application. Clearly, searchable encryption has the potential to provide enhanced security should the necessary operations, such as updates, be efficiently and securely supported.

Disjunctive queries also allow to perform range queries. Shi et al. first demonstrate this in [27]. Lu shows that one can also construct an index for range queries [23].

Although searchable encryption is a clear security advantage compared to deterministic encryption, Islam et al. demonstrate an attack based on the leaked access patterns [18]. Their attack relies on the knowledge of the distribution of keywords and works for many ciphertexts with a corresponding search token. Hence, it is even more important to not leak additional information during updates.

9. CONCLUSIONS

We have demonstrated a new technique for efficient, dynamic searchable encryption. Our idea is to learn the index from the search token. We have theoretically shown that this must lead to the optimal search time over a sufficiently long period. We have experimentally shown that this search time is low in absolute numbers and hence highly practical. Furthermore, it is reached much faster in practice than the theoretical upper bound.

Our scheme can be implemented without client storage and only requires to store 2 cryptographic hash values per index entry. Additions and deletes can be performed securely. We maintain semantic security even during updates, i.e. we leak no additional information. We give detailed leakage functions in our simulation-based security proof that compare favourably with related work. In our experiments using real-world search terms, 84% of all keywords were never searched for. These keywords remain semantically secure encrypted and hence profit from the additional security under updates provided by our scheme.

We believe our construction to be valuable from two perspectives. First, it provides a novel design alternative for constructing dynamic searchable encryption scheme. Our idea may be also applied to other research directions of searchable encryption, such as complex queries. Second, it provides a favorable trade-off compared to deterministic encryption, since it is almost as efficient – the time for the second search of a keyword is the same –, but significantly more secure. Hence, it may provide a viable alternative for practical adoption.

Acknowledgements

We would like to thank our reviewers for their insightful comments that helped improve the paper and our shepherd Frederik Armknecht who provided many helpful hints for finishing the paper in the best possible mode. The work in this paper was supported by the European Union Seventh Framework Program (FP7/2007–2013) under grant agreement no. 609611 (PRACTICE).

10. REFERENCES

[1] `http://www.gutenberg.org/`.

[2] `http://invokeit.wordpress.com/frequency-word-lists/`.

[3] `http://www.ciphercloud.com/`.

[4] `http://www.vaultive.com/`.

[5] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In *Advances in Cryptology*, CRYPTO, 2007.

[6] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology*, EUROCRYPT, 2004.

[7] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *Proceedings of the 4th Theory of Cryptography Conference*, TCC, 2007.

[8] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Proceedings of the 21st Network and Distributed System Security Symposium*, NDSS, 2014.

[9] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Proceedings of the 33rd Cryptology Conference*, CRYPTO, 2013.

[10] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6), 1998.

[11] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5), 2011.

[12] C. Dong, G. Russello, and N. Dulay. Shared and searchable encrypted data for untrusted servers. *Journal of Computer Security*, 19(3), 2011.

[13] E.-J. Goh. Secure indexes. Technical Report 216, IACR Cryptology ePrint Archive, 2003.

[14] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3), 1996.

[15] P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In *Proccedings of the International Conference on Applied Cryptography and Network Security*, ACNS, 2004.

[16] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM International Conference on Management of Data*, SIGMOD, 2002.

[17] H. Hacigümüs, S. Mehrotra, and B. R. Iyer. Providing database as a service. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE, 2002.

[18] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS, 2012.

[19] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Proceedings of the 17th International Conference on Financial Cryptography and Data Security*, FC, 2013.

[20] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS, 2012.

[21] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Advances in Cryptology*, EUROCRYPT, 2008.

[22] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th IEEE*

Symposium on Foundations of Computer Science, FOCS, 1997.

[23] Y. Lu. Privacy-preserving logarithmic-time search on encrypted data in cloud. In *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS, 2012.

[24] M. Naveed, M. Prabhakaran, and C. Gunter. Dynamic searchable encryption via blind storage. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[25] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP, 2011.

[26] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using mylar. In *Proceedings of the 11th USENIX Symposium of Networked Systems Design and Implementation*, NSDI, 2014.

[27] E. Shi, J. Bethencourt, H. T.-H. Chan, D. X. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *Proceedings of the 2007 Symposium on Security and Privacy*, S&P, 2007.

[28] R. Sion and B. Carbunar. On the practicality of private information retrieval. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS, 2007.

[29] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy*, S&P, 2000.

[30] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable symmetric encryption with small leakage. In *Proceedings of the 21st Network and Distributed System Security Symposium*, NDSS, 2014.

[31] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS, 2013.

[32] P. van Liesdonk, S. Sedghi, J. Doumen, P. H. Hartel, and W. Jonker. Computationally efficient searchable symmetric encryption. In *Proceedings of the 7th VLDB Workshop on Secure Data Management*, SDM, 2010.

[33] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *Proceedigs of the 19th ACM Conference on Computer and Communications Security*, CCS, 2012.