This is a post-peer-review, pre-copyedit version of an article published in the International Journal of Information Security. The final authenticated version is available online at: https://doi.org/10.1007/s10207-018-0414-4

Dynamic Searchable Symmetric Encryption for Storing Geospatial Data in the Cloud

Benedikt Hiemenz · Michel Krämer

Abstract We present a Dynamic Searchable Symmetric Encryption scheme allowing users to securely store geospatial data in the cloud. Geospatial data sets often contain sensitive information, for example, about urban infrastructures. Since clouds are usually provided by third parties, these data needs to be protected. Our approach allows users to encrypt their data in the cloud and make them searchable at the same time. It does not require an initialization phase, which enables users to dynamically add new data and remove existing records. We design multiple protocols differing in their level of security and performance respectively. All of them support queries containing boolean expressions, as well as geospatial queries based on bounding boxes, for example. Our findings indicate that although the search in encrypted data requires more runtime than in unencrypted data, our approach is still suitable for real-world applications. We focus on geospatial data storage, but our approach can also be applied to applications from other areas dealing with keyword-based searches in encrypted data. We conclude the paper with a discussion on the benefits and drawbacks of our approach.

Keywords Cryptography · Private Information Retrieval · Geographic Information Systems · Cloud Computing

B. Hiemenz · M. Krämer
Technische Universität Darmstadt, Darmstadt, Germany

M. Krämer
Fraunhofer Institute for Computer Graphics Research IGD
Darmstadt, Germany
Tel.: +49-6151-155 415
Fax: +49-6151-155 444
E-mail: michel.kraemer@igd.fraunhofer.de

1 Introduction

In recent years, more and more companies have started to outsource data and computations to the cloud. They expect many benefits from doing so. A cloud infrastructure allows for a worldwide data access and other benefits such as scalability and elasticity. Such an infrastructure is mostly provided by third parties. This is a big economic advantage for many companies which can intermittently adjust their storage requirements without further hardware costs. Besides that, cloud providers often offer computation time with which customers are able to deploy and run their products in the distributed environment of the cloud provider. This enables a flexible resource management because companies can again scale the offered services at any time. By outsourcing data and computations, companies also partially hand over their responsibilities to the cloud provider which (depending on the contract) is put in charge of important aspects such as backup management and availability.

On the downside, companies lose control over their own data by outsourcing them. Cloud providers usually have full access to data stored in their infrastructure. Moreover, many cloud providers have several data centers around the world. Since companies are not always allowed to choose where their data will be stored, they may face problems with local law regulations. European companies, for example, are bound to EU law and must not outsource confidential data like personal ones to data centers outside the EU without ensuring that the foreign cloud provider complies with EU principles (see Article 45, EU-GDPR [5]).

A common way to secure data in an untrustworthy system (e.g. one provided by a foreign cloud provider) is to apply cryptographical protection such as encryption. However, encryption negates many advan-

tages that have been advertised by cloud providers in the first place. Most forms of computation are much more difficult to accomplish if they have to operate on encrypted data. The storage itself causes problems, too. In case their data are encrypted, owners are no longer able to search them. Queries on encrypted data are challenging if you do not want to break the encryption or run the search locally on the owner's side. In addition, the query itself can leak sensitive information about the data.

Searching encrypted data without leaking the query is a growing field in cryptography. Several approaches to this topic have been published in the last decades. The most promising one is called Searchable Symmetric Encryption (SSE), because it is the only existing approach achieving runtimes that are suitable for real-world applications. SSE allows data owners to make their encrypted documents searchable. This does not include a full-text search but works on a keyword-based technique. Owners tag their documents with any number of keywords and store all associations in an index. Later on, they are able to search for certain documents based on their associated keywords. The index and the documents can be stored safely in the cloud as both are encrypted. Not only the index and documents are secured, but also the query leaks minimum information. Use cases for SSE exist in many areas. A simple but popular example are emails since they are almost always stored on the provider's infrastructure nowadays. Emails often contain confidential information. They should be secured but remain searchable at the same time. SSE can be a reasonable solution in such a scenario.

In this paper, we are not focusing on emails but another use case: geospatial data. This type of data describes regions, urban areas, etc. and can include a high level of detail such as information about streets or buildings. Depending on the project, these data are confidential and must be secured before they are outsourced to the cloud. Geospatial file stores are optimized for this type of data. An example for such an application is GeoRocket [8]. It is optimized for geospatial files, provides high-performance data storage, and supports several cloud infrastructures as back-end storage (such as Amazon S3). All data in GeoRocket are stored in plaintext. This setting has to be improved to provide a suitable environment for confidential data.

1.1 Contribution

Making encrypted data searchable is not bound to a particular data type, but knowledge about the file's structure can be an advantage as we will see in the course of this paper. Our design adapts some techniques from existing SSE approaches [11, 17] and is inspired by previous work from Cash et al. [1, 2] but is specifically optimized towards our requirements for geospatial data storage (i.e. structured data including attributes that need to be searchable) and the spatial queries we need to perform. The way we apply SSE to cloud-based data storage based on structured files that can be split into chunks, in combination with the kind of queries our system supports, is novel.

We focus on performance and usability, which means our search is fast and transparent to the user. Our evaluation shows that we leak more information compared to related projects and thereby our approach is more vulnerable to certain kinds of attacks such as statistical ones. We define this leakage and describe its consequences. The leakage is acceptable to us because of two reasons. Our system provides enough security to resist several kinds of attacks and is hence suitable for many scenarios. Furthermore, we demonstrate that alterations may provide a higher level of security but are only temporary and dramatically increase the search time or reduce the usability. Our SSE system supports parallel processing which improves the performance. On the client side, we assume nothing more than cryptographic keys. Therefore, our approach facilitates multi-device support and can easily be ported to clients with limited memory. We support boolean expressions and range queries. The latter are used to search for geospatial data using *bounding boxes* (an area specified by four coordinates $minX$, $minY$, $maxX$ and $maxY$).

SSE is based on an inverted index and in this paper, we particularly focus on index encryption. The actual geospatial data are stored in the cloud using symmetric encryption with a traditional cipher scheme such as AES.

1.2 Outline

The structure of this paper is as follows: Section 2 summarizes research related to our work. We compare our approach to others and explain why our techniques are more applicable regarding our requirements. Some of these works use SSE, but others achieve searchable encryption with different approaches.

Section 3 describes GeoRocket, the software we implement our SSE system in.

In Section 4 we introduce the concept of our SSE system. In Section 5 we focus on index security and explain how users can create an encrypted database with keyword/document association and run private queries against it to search their data.

After the theoretical part we give an overview of our implementation work in Section 6. Section 7 evaluates

our implementation. We present several benchmarks to report on performance and to compare our operations with their unencrypted counterparts. We conclude this paper in Section 8 and give an outlook on future work.

2 Related Work

This section outlines research related to the work done in this paper. We summarize SSE-based approaches and other techniques handling private queries on encrypted data.

2.1 SSE-based Approaches

The first paper introducing SSE was published by Song, Wagner and Perrig [16]. The classification of an SSE system is not clearly defined. Techniques that exclusively combine symmetric encryption with search capabilities can be summarized as SSE. All systems implementing SSE share some characteristics. They leak the access and search pattern (see Section 5.1). No obfuscation is applied to hide which (and how many) encrypted documents are part of a search outcome. This does not leak any information about the documents' content but can facilitate an inference to the queried keyword. The server learns how frequent the keyword is in relation to the entire index. Leaking this allows SSE protocols to deliver the search result to the client in only one communication step. In addition, the server is able to detect recurring queries, because the query is obfuscated deterministically. These weaker security assumptions are one reason why SSE provides such a good performance in comparison to other approaches, which on the other hand may provide more privacy. Additionally, SSE techniques accomplish search functionality by creating an index including keyword/documents associations. No SSE system operates on the encrypted data directly. To provide a full-text search, all document's words must be part of the index.

In literature, SSE is mostly divided into two approaches: static and dynamic. Static SSE schemes neither support the adding of new documents nor the deletion of existing records. An update operation is usually not possible, too. After initialization, the index is frozen. As a consequence, users must be in possession of all documents on startup. This is reasonable in many scenarios but unacceptable for us. Like most geospatial file stores, GeoRocket is intended to provide storage over a long period and it has to be possible to add or delete documents at any time. In contrast to static schemes, dynamic ones support adding and deleting.

An SSE index is usually built using key/value data structures. A different approach is presented by Kamara and Papamanthou [11] who organize the index in a tree structure. Their search runs in sublinear time and works in parallel. Users can add more documents because their index tree is updatable. This is comparable to our approach. Boolean expressions have also been considered in their paper although their concept is in an early stage of development. We support boolean expressions. But the work of Kamara and Papamanthou has one crucial limitation: the universe of keywords must be known at the beginning and cannot be extended later. Documents can be easily added but only associated with keywords which have been defined by the user on startup. Therefore, users must either define a huge set of keywords or guess which keywords could be interesting in future documents. Our approach does not have this limitation as we allow for an arbitrary amount of keywords and can handle unseen ones easily.

Stefanov, Papamanthou and Shi focus on the process of adding documents [17]. All dynamic techniques must deal with a problem leading to information leakage: if a single document and its keywords are added after an initialization phase, it will be obvious for the honest-but-curious server that these new index entries belong to the new document. We prevent this by taking advantage of GeoRocket's splitting process (see Section 3). The GeoRocket Server can only link the index entries to the entire file not to single chunks. Stefanov, Papamanthou and Shi also present a general solution how this leakage can be avoided. Their approach however is not suitable for us. Client and server run an extra protocol after each add operation in which the client downloads parts of the index, re-encrypts it and sends it back. Shuffling the index results in a fresh state, where the server cannot identify the recently added associations anymore. However, their protocol is time-consuming and requires both parties (server and client) to stay connected the whole time. It is not user-friendly to forbid the client to close its session after the actual add operation is done. They also do not support boolean expressions but only the search for a single keyword.

Most papers dealing with SSE focus on two kinds of index handling. Either all keywords are stored to a particular document (index) or all documents are stored to a particular keyword (inverted index). Since SSE provides a keyword-based search, the latter is faster. We use an inverted index. A regular index usually causes linear complexity, because the server must hit each document entry to check if it contains the queried keyword. An inverted index does not need that many hits and can be sublinear or even optimal/constant. However, the se-

curity can be stronger in a regular index as shown by Yavuz and Guajardo [19]. Their approach is dynamic, provides strong security but only linear runtime, which is unsuitable for huge data sets.

A combination of both indexing techniques has been introduced by Hahn and Kerschbaum [10]. Their approach is based on the observation that the access pattern is leaked once a keyword was part of a query. The stronger security provided by a regular index is therefore only temporal. They have designed a protocol where the server stores the index regularly at first. Searches are analyzed and (if necessary) parts of the index are gradually rebuilt changing to an inverted representation. The runtime decreases with each search. We decided to use an inverted index from the beginning since the security improvement is only temporal and rebuilding the index is memory and time consuming.

2.2 Other Approaches

Beyond researches about SSE, a few more approaches are applicable to search encrypted data. Examples are fully homomorphic encryption and oblivious random-access memory. Both are not limited to searching but can be seen as general-purpose solutions.

Fully homomorphic encryption (FHE) was first published by Gentry [7]. His work describes a system allowing everyone to operate on a ciphertext without knowing the secret key. An example is multiplication: $Enc(a) * Enc(b) = Enc(a * b)$. FHE is not limited to a single operation but supports arbitrary ones. Users can upload encrypted data to a cloud provider, which can perform any operation on the data without decrypting them. This makes FHE a promising approach to enable full-text searches on encrypted data. Unfortunately, FHE requires a lot of performance and runtime and is yet not fast enough for real-world applications.

Another technique to search encrypted data is oblivious random-access memory (ORAM), which was first proposed by Goldreich [9] and later revised by Ostrovsky [14]. ORAM describes a system where a server provides storage and allows clients to read and write data without knowing which location within the store is being accessed by the client. This is achieved by shuffling and re-encrypting the data periodically or after each access. Like FHE, ORAM is not limited to searches in encrypted data but can also be used for other tasks. ORAM provides a much higher level of privacy than SSE as nothing but the overall storage size is leaked. However, the technique is not practical because of the long runtime the encryption and searching operations require.

3 GeoRocket

To evaluate our SSE system, we implement it as an extension to GeoRocket [8]. GeoRocket is a data store for geospatial files, which is able to store 3D city models (e.g. CityGML), GML files or any other data sets in the formats XML or GeoJSON. It can be configured to use the local file system as storage back-end or to access a distributed service such as Amazon S3, MongoDB, HDFS, or Ceph. Although each back-end has advantages, privacy could be an issue in case an external data store service is used because the provider might not be trustworthy. GeoRocket is hence a good example for the aspects we described in our introduction.

GeoRocket has an asynchronous, reactive and scalable software architecture, which is depicted in Figure 1. The import process starts in the upper left corner. Every imported file is first split into individual *chunks*. Depending on the input format, chunks have different meanings. 3D city models stored in the CityGML format, for example, are split into cityObjectMember objects which are typically individual buildings or other urban objects. The data store keeps unprocessed chunks. This enables users to later retrieve the original file they put into GeoRocket without losing any information.

Attached to each chunk is metadata containing additional information describing the chunk. This includes tags specified by the client during the import, automatically generated attributes and geospatial-specific ones

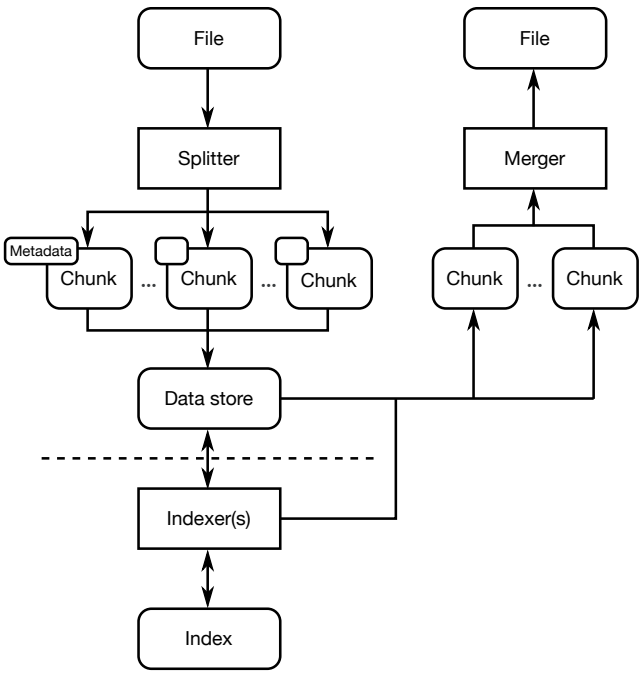

Fig. 1 The software architecture of GeoRocket

such as bounding boxes or the spatial reference system (SRS). Immediately after the chunks are put into the GeoRocket data store, the indexer starts working asynchronously in the background. It reads new chunks from the data store and analyses them for known patterns. It recognizes spatial coordinates, attributes and other content.

The export process starts with querying the indexer for chunks matching the criteria supplied by the client. These chunks are then retrieved from the data store (together with their metadata) and merged into a result file.

The index is maintained by Elasticsearch [4]. Indexed data are searchable using a flexible query language. Users can combine any number of keywords with `AND`, `OR`, `NOT` terms allowing them to create nested queries. Some special expressions are also supported. Four numbers separated by a comma, for example, are compiled to a query specifying a bounding box. The following string is compiled to a query for all chunks containing the attribute (or tag) `Building` located inside the given bounding box (denoted by four numbers):

```
AND(Building 13.378,52.515,13.380,52.517)
```

Several operators can be combined to build more complex queries such as the following one:

```
AND(OR(Building Tree)
    NOT(13.378,52.515,13.380,52.517))
```

This query would result in all chunks containing the attributes (or tags) `Building` or `Tree` but which are *not* located in the given bounding box. The default operator is `OR`. If no operator is specified GeoRocket will search for all chunks matching *any* of the given criteria.

The application is written in Java and includes two parts, *GeoRocket Server* and *GeoRocket CLI*. All tasks described above are handled by the server. Once data are imported, the server splits, stores and indexes them. Furthermore, queries are executed on the server side to find and return chunks that match the search criteria. The other component is a Command-Line Interface (CLI) that runs on the client side. Using the CLI, users can interact with the server and execute commands to import, export, search and delete files or chunks.

4 Searchable Symmetric Encryption

As mentioned before, SSE systems are characterized by symmetric encryption combined with search capabilities. All of them have a similar procedure. First, users create an index of all documents they want to search for later. The index contains keyword/document associations showing which documents and keywords are linked. Both are obfuscated in some way. In addition, users encrypt the documents with a traditional cipher scheme such as AES. The index and documents can now be sent to the cloud. Later on, users are able to run a private search including the keyword they are interested in. Querying the index results in a collection of document identifiers indicating which of the encrypted documents matches the queried keyword. Finally, the users receive these encrypted documents and decrypt them locally.

Our SSE system involves a client and a server. We assume that the client is controlled by a user and entirely trustworthy. The server is honest-but-curious and located in the cloud. Our index uses a key/value data structure to store pairs including the keyword as key and a list of document identifiers as the corresponding value. Figure 2 shows the components of our SSE system. The client interacts with the server, which stores an encrypted index and a collection of encrypted documents. The documents are linked to their corresponding index entries via an identifier. In contrast to most SSE approaches, we do not distinguish between an initialization step and following operations. We assume an empty store as a starting point and allow users to add documents at any time. Our security assumption does not depend on the time a document is added. The first document is as secure as the second, third, etc.

Depending on the environment, it can be beneficial to address the encryption of documents and the index separately, for example, if users store their encrypted documents at one cloud platform, and store and query the index on another. The cloud platform for the documents can be optimized for data storage and the other to support computations and to perform queries. In case the index and documents are maintained sep-

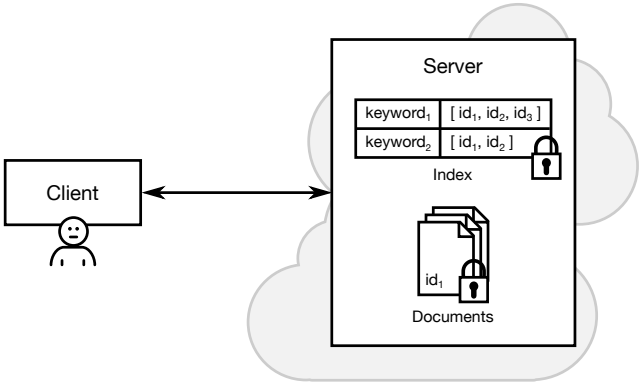

Fig. 2 Involved parties in SSE: client and server storing an encrypted index and encrypted documents

arately by different providers that do not share information with each other, the security will increase, too. Both providers interact with the client only and do not need any information about each other. The server responsible for the document storage does not need to know anything about keywords. The server responsible for the index does not need to know anything about the documents but their identifiers. Our protocol does not specify whether these two tasks are fulfilled by one or two cloud providers. If the protocol is running between a client and two providers, both are considered honest-but-curious.

Static SSE systems benefit from the fact that all documents must be available on startup. Adding a new document to the index leaks some information about it—for example the number of keywords the new document is associated with. If the server retrieves only one document, it is obvious that all keywords received in the same add operation must belong to this document. Static techniques are not affected by that, because the server gets all keyword/documents associations at once and (in most protocols) cannot distinguish between particular documents. The only leakage is the sum of documents and keywords. We have multiple options to avoid this in dynamic approaches. This leakage is weakened in case the client sends more than one document at the same time. Most protocols—including ours—support this. The client is responsible to shuffle the new index entries before sending them to the server. In case two documents are added in the same process, the server should not assume that the first part of the index entries belongs to the first received document. This technique expects the client to always send multiple documents and makes the security depend on it. The more documents are added in one process the less obvious it is for the server to detect which keywords belong to which document.

Another option is to generate random keywords and use them as fake entries. This prevents the server from determining the number of keywords associated to a document because only the client can distinguish between a fake and a real keyword. This solution causes further issues like how many fake entries should be generated and how much storage and performance is wasted because of that.

We can take advantage from our focus on geospatial files and GeoRocket's import process which splits structured files into separate chunks (e.g. a file containing a 3D city model is split into individual buildings). Although users import files to GeoRocket, the chunks are stored and not the original file. Our keyword-based search focuses on chunks, too. A search result contains all chunks matching the queried keyword merged into a single file. The keywords are mostly generated by the GeoRocket indexing process which operates on each chunk separately. Only user-specific tags are associated with all chunks. Therefore, when we talk about documents, we do not talk about files but about chunks.

Our add operation automatically includes several documents, because geospatial files are likely to consist of multiple chunks. The server only learns how many keywords are associated with the entire file, not with a particular chunk.

In the next section, we introduce a technique to secure the index and support complex but private queries which do not leak information about keywords. Our approach is evaluated afterwards and discussed in the context of several extensions which provide a higher security but increase the runtime and impact usability.

5 Index Security

We start this section with some security definitions that are important in the context of SSE (see Section 5.1). Afterwards we focus on single keyword search exclusively. We describe our protocol, which does not include an initialization but starts with an empty data store and index. Users are able to add documents and tag them with an arbitrary number of keywords. Additionally, we offer search and delete operations. Both are working keyword-based which means that users specify a single keyword in a request to get or delete all documents that are associated with the queried keyword. We present two protocols supporting single keyword search. The first one—named the *basic protocol*—leaks more information to the server but achieves a good performance. It is described in Section 5.2. The second protocol improves the security, but the runtime of all operations increases. We present this *extended protocol* in Section 5.3.

We also cover boolean expressions as we embed the Basic Cross-Tags (BXT) and the Oblivious Cross-Tags (OXT) protocol [1] in Section 5.4. This allows us to support the operators AND and NOT. We focus on the add and search operations. The delete operation of BXT and OXT is beyond the scope of this paper. Although the main part of our protocol is not exclusively designed for GeoRocket, we sometimes take advantage of certain GeoRocket characteristics or discuss a limitation regarding the software.

Note that in this paper we focus on describing an approach for encrypting the index. The actual chunks (or documents) stored in the cloud are encrypted symmetrically using a traditional cipher scheme such as AES.

5.1 Security Definitions

The more information is leaked the more the honest-but-curious server is able to draw conclusions about the plaintext of either the data or the query or both. However, not every leakage causes a comprehensive security break. Sometimes it can be acceptable to reduce the security in order to improve other aspects such as the performance of a system. It is important to know what information is leaked, who could get access to it and what the consequences could be. The following definitions are widely used, not only in the context of SSE but also for other techniques achieving searchable encryption.

Search Pattern. Leaking the search pattern allows the server to detect recurring keyword searches. Although the server is not able to learn the keyword (because it is usually obfuscated), the query can be saved and compared with queries in the future. If the server knows that a client has been searching for the same keyword in earlier queries, the search pattern is leaked. SSE schemes (including ours) usually leak the search pattern.

Access Pattern. The access pattern describes the leakage of a search outcome. After a query has been executed, the server is able to learn how many and which (encrypted) documents are part of the response and therefore that theses documents are associated with the queried keyword. This information is not leaked at the time of adding new documents but as soon as the client searches for them. If this kind of information can be learned by the server, the access pattern is leaked. The location of documents the client wants to access is not secured against an honest-but-curious server.

Forward Privacy. The moment a query (consisting of a single keyword) is executed, the server learns which documents are associated to that particular keyword (access pattern). This information can be extracted by the search outcome. Assuming a user has run several searches in the past, forward privacy is ensured if the server does not know that a newly added document is associated with keywords which have been part of a query in the past. A system provides forward privacy in case the server cannot relate a keyword of a new document with anything known. Even if this particular keyword already exists in the index and even if the user has searched for it before.

Backward Privacy. Backward privacy concerns documents being deleted instead of newly added ones. In case a query (consisting of a single keyword) is executed, backward privacy is ensured if the server does not know that the queried keyword is associated with documents which are already deleted. At the time of searching, the server does not learn anything in retrospect about the keyword association of documents that are no longer part of the index.

5.2 Basic Protocol

We start with an SSE protocol covering single keyword searches only—we add support for boolean expressions later (Section 5.4). We describe the protocol steps client and server must follow to make encrypted documents searchable. Our approach adapts a few concepts from Cash et al. [2]. However, both techniques are only partly comparable. Their approach provides an initialization phase and distinguishes between documents added in the setup phase and those added later on. Security and performance can therefore differ from document to document. We do not make such distinction because we do not assume users are in possession of all documents on startup. The supported features also differ as we support range queries (for bounding boxes) and they do not. However, both approaches use a similar obfuscation technique to mask index entries (except coordinates referring to range queries).

We support three operations: add, search and delete. All of them require interaction between the client and server. Several cryptographic keys are required on the client side to perform these operations. The server does not need any keys.

5.2.1 Add Operation

Our store is empty at first and we do not have an initialization step. Users are able to add documents immediately and constantly. The client reads one or more files as input and creates two collections. One includes all encrypted documents the other one contains the encrypted index entries. The server receives the two collections from the client.

As first step of the add operation, the client iterates through the file to be added and extracts its chunks. Each chunk represents a document in the context of SSE. In the following, we use the term *document* for a consistent terminology. The client generates a unique identifier for each document, because not the file but documents must be addressable. This identifier must not be related to the document's content as the server learns it later on. The client encrypts the document

```
Key k_AES, k_PRF, k_OPE
Collection indexEntries
Collection documents

for each document in file, do:
    documentID = Identifier()

    // encrypt document
    encDocument = Enc(k_AES, document)
    encDocument.attach(documentID)
    documents.add(encDocument)

    // index security
    for each keyword, do:
        if keyword is of type bounding box
            obfuscatedKeyword = (OPE(k_OPE, minX), OPE(k_OPE, minY), OPE(k_OPE, maxX),
                OPE(k_OPE, maxY))
        else
            obfuscatedKeyword = PRF(k_PRF, keyword)

        encryptedID = Enc(k_AES, documentID)
        indexEntries.append(obfuscatedKeyword, encryptedID)

indexEntries.sort()
sendToCloud(documents)
sendToCloud(indexEntries)
```

Listing 1 Single keyword search: add operation on the client side. Basic protocol.

with traditional cipher schemes such as AES. In addition, the identifier is attached to the encrypted document. The client then puts the ciphertext into the corresponding collection which will be sent to the server in the end.

The next step concerns the document's keywords. For our protocol, it does not matter whether a keyword has been defined by the user or if the client selects special words from the document as such. We assume that a set of keywords exists for each document. The client iterates through this set and generates a pair for each one. One value is an obfuscation of the keyword itself, the other one is the encrypted identifier of the document currently handled. The obfuscation technique depends on the keyword's type. We use a pseudorandom function (PRF) with a cryptographic key and the keyword as input. Because of its underlying cryptographic characteristics, the PRF output is indistinguishable from a random string and the server cannot reconstruct the keyword. Additionally, performing a PRF provides a good performance as it is not time-consuming.

To support range queries (for bounding boxes), we need a special handling of keywords that represent coordinates. The obfuscation technique must allow the client to preserve the numerical order of the coordinates so the server is able to handle range queries. A few techniques are suitable to deal with this issue. We use OPE to support range queries. OPE is an symmetric encryption scheme preserving the numerical order of the plaintext. It is defined as $\forall a, b \in \mathbb{N} : a < b \rightarrow f(a) < f(b)$ with an OPE function f. The codomain of f must be greater than its domain so $\forall A, B \in \mathbb{N} : \quad |A| < |B| \quad$ for $\quad f : A \rightarrow B$. By preserving the order, OPE supports efficient range queries on encrypted data. The keyword obfuscation is always the same because PRF and OPE work deterministically and we do not append any random input.

To encrypt the document identifier for the index, the client uses traditional cipher schemes such as AES. Because we generate a fresh Initialization Vector (IV) each time, the resulting ciphertexts vary even if the same plaintext is encrypted twice. Therefore, ciphertexts are unique and multiple of them can refer to the same identifier. The pairs of keyword/identifier (both obfuscated or encrypted) are put into another collection.

After all documents have been traversed, the client holds two collections. One includes all keyword/documents associations, another stores the encrypted documents (with the corresponding identifier attached in plaintext). The former collection must be sorted lexicographically to hide the order in which the keywords have been added. In the end, two collections are sent to the cloud. On the server side, the add operation does not need much effort. The server receives both collections and stores them. If two providers are used in this

protocol, the client sends the collection respectively to their responsibilities. Table 1 shows an index with six documents and three keywords (including one bounding box consisting of four numbers $minX$, $minY$, $maxX$, and $maxY$). As mentioned before, each Enc(id_i) differs even for the same i. The keywords are obfuscated deterministically and can therefore be merged.

Obfuscated Keyword	**Encrypted Identifier**
PRF($keyword_1$)	Enc(id_1), Enc(id_2), Enc(id_3), Enc(id_4)
PRF($keyword_2$)	Enc(id_2), Enc(id_4), Enc(id_5)
(OPE($minX$), OPE($minY$), OPE($maxX$), OPE($maxY$))	Enc(id_6)

Table 1 Encrypted index

Our add operation requires only one communication round and is time-consuming on the client side. We require some cryptographic transformations, including the document encryption and the PRF or OPE generation for the keywords. Because the data is encrypted we cannot outsource these operations to the server, which would actually be responsible for building the index.

The client needs three different keys for the add operation. One for the AES encryption, another for OPE and the last for PRF. If more files should be added at once, the documents and index entries can be sent as a joint request respectively. It is not necessary to create two new collections for each file. Using only one instance of both is highly recommended because it prevents the server from distinguishing which keyword/documents association belongs to which file. The pseudo-code in Listing 1 illustrates our protocol to add documents to the store and summarizes the steps above.

5.2.2 Search Operation

Figure 3 illustrates our search operation. We only support the search for a single keyword for now. The client regenerates the PRF result of the keyword to search for. Since PRFs are deterministic as long as the same key and keyword are used, the result matches the index entry on the server side (if the keyword exists in the index). The client sends the PRF output to the server ❶. If an entry is found, the server responds with a list of encrypted identifiers corresponding to the PRF output ❷. The client decrypts them and filters out duplicates (in case some encrypted identifiers refer to the same plaintext). The resulting set of identifiers is sent back to the server ❸ which then returns the documents ❹. Since these are still encrypted, the client decrypts them as the last step. At the end, the client has all documents associated with the queried keyword.

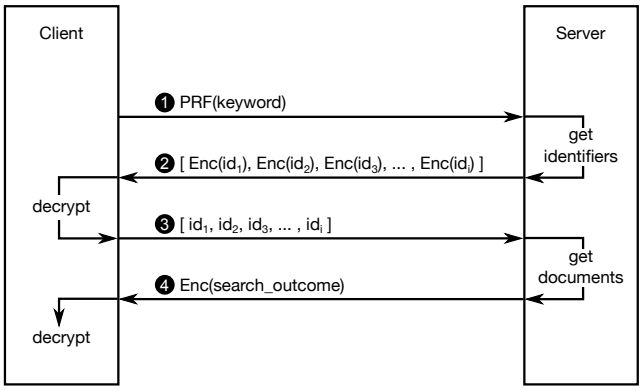

Fig. 3 SSE search protocol between client and server

To run range queries, the client encrypts the desired coordinates with OPE and sends the ciphertext to the server. Since OPE works deterministically, the ciphertext is the same as during the add operation. To query a valid bounding box, the client must specify all four coordinates ($minX$, $minY$, $maxX$, $maxY$). Furthermore, the client sends information whether it is interested in documents whose bounding boxes are intersecting with its input or not. Due to the order preserving, the server can look up all ciphertexts whose bounding box matches the search criteria by comparing whether the coordinates are respectively lower or higher.

Unlike our add operation, searching requires two communication rounds. The server cannot handle the encrypted identifiers, only the client is able to work on them. This increases our security because the server cannot draw inferences from identifiers to their obfuscated keywords. For example, the server cannot count the number of (obfuscated) keywords each document is associated with.

5.2.3 Delete Operation

Our last operation is similar to our search and handles the deletion of documents. We provide a keyword-oriented delete operation hence users cannot remove a certain document but all documents related to a single keyword. At first, the client generates the keyword's PRF result and sends it to the server. If an entry to this value exists in the index, the server responds with a list of corresponding encrypted identifiers. Additionally, the server deletes the row in the index. The client receives the list and decrypts the identifiers. Finally, it makes a request to the server instructing it to delete these documents from the store. Deleting documents based on an OPE query works similarly.

Our deletion is not complete as some pieces of data remain on the index. Table 2 shows the problem.

Obfuscated Keyword	Encrypted Identifier
~~PRF($keyword_1$)~~	~~Enc(id_1), Enc(id_2), Enc(id_3), Enc(id_4)~~
PRF($keyword_2$)	**Enc(id_2)**, **Enc(id_4)**, Enc(id_5)
(OPE($minX$), OPE($minY$), OPE($maxX$), OPE($maxY$))	Enc(id_6)

Table 2 Encrypted index after a delete operation. Deleted entries are crossed out. Orphaned identifiers are highlighted.

If the client sends PRF($keyword_1$) to the server as part of a deletion request, the corresponding row is deleted from the index as well as documents id_1, id_2, id_3, id_4 from the store. However, the row of the obfuscated keyword PRF($keyword_2$) still contains ciphertexts whose respective plaintext refers to documents that do not exist anymore (orphaned id_2 and id_4). This affects only the index not the document storage. On the next search request for PRF($keyword_2$), the server responds with Enc(id_2), Enc(id_4), Enc(id_5) not knowing that id_2 and id_4 have been deleted in the past. This does not break our procedure. The client decrypts the response and sends id_2, id_4, id_5 back to the server which replies with an error for id_2 and id_4 and hence sends only one encrypted document back. Nevertheless, the search is perfectly valid, but we wasted some traffic and cryptographic operations for outdated identifiers.

During the delete operation, we cannot remove all encrypted identifiers correctly without a huge performance loss. The server would have to send the entire index so the client can decrypt all entries and mark every ciphertext referring to a document to be deleted. This is neither scalable nor practical. A more promising option is cleaning the index gradually. Whenever a search is running, the client caches the encrypted identifiers it has received from the server. If the server then indicates that one of the requested documents has been deleted, the client is able to show the server which entries can be removed within the index. This way, we do not delete everything directly but distribute the load of the client to several steps over time. One more communication round is required for our search operation as the client sends a list of outdated encrypted identifiers to the server after a query is complete.

5.2.4 Discussion

Our protocol is optimized to achieve a good performance. Due to our inverted index, the server must hit the index only once to look up all entries for a keyword. The used cryptographic operations are known for their low runtime and are thus practical. We require one communication round to add documents and two for a delete or search operation (three in case of our gradually index cleaning). Except update capabilities, we support everything a dynamic SSE approach requires. Therefore, users do not have to be in possession of all documents on startup but can add and delete records at any time. Our SSE system does not support multi-reader or multi-writer features because only the client is in possession of the keys. Read and write permissions are therefore exclusively permitted to the owner.

Besides the search for words, our protocol supports range queries allowing the client to run bounding box-related query. In addition, our protocol allows for parallel execution. Some SSE approaches (such as [12]) use a linked list as data structure for the index security. This technique may have advantages but also forces a sequential process as the entries can only be accessed one after another. Because our data access is not bound by a linked list, we do not suffer from a sequential procedure. The system is user-friendly as our protocol runs transparently to the user. No operation requires an extra step users would have to care about. The keys can be generated and handled in the background, no settings require the user's attention. The client is allowed to close the session immediately after an operation is done. Neither keys nor plaintext leave the client's system. On the client side, we require a keystore (including three keys) only. No further storage is needed. Our protocol is therefore suitable regarding multi-device support. Users must synchronize their devices only once because the keystore does not change after all keys have been generated.

On the other hand, we achieve our performance at the expense of security. The client encrypts the document identifiers, so the server is not able to learn if two keywords are associated with the same document. Taking an encrypted identifier as starting point, the server observes no further information, neither if the document is associated with another keyword nor how many keywords it has. Preventing the server from counting is a good defense against statistical attacks because we expect the server to be honest-but-curious. Taking a keyword as starting point, the server however is able to learn significantly more, which can be shown by our prior security definitions.

The access pattern is leaked if the server is able to associate the queried keyword with the search outcome. This leakage depends on the environment. Our protocol contains two communication rounds. The server receives the queried keyword in one request. However, the document identifiers are part of a different request. Leaking the access pattern depends on the server's capability to link these two requests. As mentioned before, our protocol allows the client to interact with two servers. One is responsible for the index and the other

is responsible for the document storage. Under the assumption that both servers do not share information with each other, linking the requests is not likely. Therefore, the access pattern is not leaked in such a setup. But in general, our protocol does not ensure that this kind of leakage is prevented.

Forward privacy is not achieved, because all identifiers associated with a certain keyword are stored in the same row. The server knows which entries were hit in the past. If a new document is added, the server learns whether it is associated to an existing keyword or not. The main problem regarding forward privacy is that our keyword obfuscation works deterministically. Using this setting, a particular keyword can be easily tracked by the server.

Since we fail forward privacy, we allow the server to count how many documents are associated with a certain keyword. The server sums the entries in a particular row to get this information. We prevent the server from counting how many keywords a particular document has, but not the opposite direction. Counting allows the server to conclude the popularity of keywords, which facilitates statistical attacks.

Backward security fails for the same reason. Again, all identifiers to a certain keyword are stored in the same row. The server is able to detect whether a deleted identifier was associated with the keyword currently queried or not. This only involves the ciphertext of identifiers not their plaintext (which is required to refer to the actual documents). However, because a delete operation works similarly to a search, the access pattern can be leaked in the process depending on the environment. In this case, the server learns which documents have been identified by the ciphertext.

5.3 Extended Protocol

This section describes an extended protocol that enables forward privacy and hence improves the security. We will discuss the extended protocol from different points of view because it also restricts usability and causes poorer performance.

As described above, the reason why our basic protocol fails forward privacy, is the fact that all encrypted identifiers are stored within the same row in the index. Using PRF is a good technique to obfuscate keywords. The client needs little effort to generate the obfuscation and the server is not able to reconstruct the keyword based on its PRF result. In addition, nothing but the key needs to be stored on the client side. PRFs work deterministically. If we want to make the output distinct each time, we must add randomness to the input.

Obfuscated Keyword	**Encrypted Identifier**
PRF($keyword_1 \parallel 0$)	Enc(id_1)
PRF($keyword_1 \parallel 1$)	Enc(id_2)
PRF($keyword_1 \parallel 2$)	Enc(id_3)
PRF($keyword_1 \parallel 3$)	Enc(id_4)
PRF($keyword_2 \parallel 0$)	Enc(id_2)
PRF($keyword_2 \parallel 1$)	Enc(id_4)
PRF($keyword_2 \parallel 2$)	Enc(id_5)

Table 3 Encrypted index with counter

However, such randomness must be stored and accessible by the client. Otherwise the client has no chance telling the server which keyword it is interested in because the client is not able to regenerate the keyword's obfuscation that has been generated during the add operation.

Our solution for this problem is inspired by the work of Cash et al. [2]. In their paper, they pointed out that only minimum information is required to obfuscate a keyword more securely. A counter is enough. PRFs are by definition not related to the input string they were generated from. Even a slight alteration of the input results in a completely different output. We make use of this and initialize a counter for each keyword. We append the counter to the keyword as part of the PRF input. Each PRF outcome differs, because the counter increases.

Our extended protocol works as follows: we initialize a counter for each keyword. Every time we create a further keyword/document pair, the counter is increased and concatenated to the current keyword. If the client searches for a keyword, it will get the current counter value i and calculate $PRF_c = PRF(keyword \| c)$ for $c = 0...i$.

Our security improves with such a counter. The server does not know how many documents are associated with a certain keyword because all rows contain only one entry now (because the obfuscation is never the same). Forward privacy is also ensured. If a new document is added, the counter increases and hides links between other index entries especially about documents referring to the same keyword.

Table 3 shows an index of our extended protocol with five documents and two keywords. We note that parts of the improvement are only temporal. At the very moment the client is searching for a certain keyword, all corresponding PRFs are generated. Sending them to the server allows for a conclusion which PRF values in the index are referring to the queried keyword. This does not affect forward privacy, because a new document is still protected by a fresh counter value.

The biggest challenge regarding our counter technique is storage. Although we do not need to save much

Obfuscated Keyword	**Encrypted Identifier**
PRF($keyword_1 \parallel 0$)	Enc(id_1)
PRF($keyword_1 \parallel 1$)	**Enc(id_2)**
PRF($keyword_1 \parallel 2$)	Enc(id_3)
PRF($keyword_1 \parallel 3$)	**Enc(id_4)**
~~PRF($keyword_2 \parallel 0$)~~	~~Enc(id_2)~~
~~PRF($keyword_2 \parallel 1$)~~	~~Enc(id_4)~~
~~PRF($keyword_2 \parallel 2$)~~	~~Enc(id_5)~~

$\mapsto$

Obfuscated Keyword	**Encrypted Identifier**
PRF($keyword_1 \parallel 0$)	Enc(id_1)
PRF($keyword_1 \parallel 1$)	Enc(id_3)

Table 4 Encrypted index with counter before and after deletion and reorganization. Deleted entries are crossed out. Orphaned entries are highlighted.

Obfuscated Keyword	**Encrypted Identifier**
PRF($keyword_1$)	Enc(4)
PRF($keyword_1 \parallel 0$)	Enc(id_1)
PRF($keyword_1 \parallel 1$)	Enc(id_2)
PRF($keyword_1 \parallel 2$)	Enc(id_3)
PRF($keyword_1 \parallel 3$)	Enc(id_4)
PRF($keyword_2$)	Enc(3)
PRF($keyword_2 \parallel 0$)	Enc(id_2)
PRF($keyword_2 \parallel 1$)	Enc(id_4)
PRF($keyword_2 \parallel 2$)	Enc(id_5)

Table 5 Encrypted index with counter stored on the server

information, we must store the counters somewhere. Storing them on the user's local system causes synchronization problems and affects our multi-device capability. Alternatively, we can store the counter on the server side. Doing so, the client receives the current counter value from the server before the search/delete or add operation is executed. If there is no counter yet, the client initializes a new one. The counter increases during the operations and must be sent back to the server in the end. Of course, the counter must be encrypted before leaving the client. To find its ciphertext again, we need some kind of identifier. For example, $PRF(keyword)$ can be used as identifier for the keyword. The counter therefore starts with 0. Using the server as counter store allows users to work on multiple devices as long as the keystore has been shared once. Overall, we need one more communication round compared to our basic protocol to get the counter.

Table 5 shows an index with two keywords, their encrypted counters and five documents. The counters are stored on the server side. We provide pseudo-code for an add operation in Listing 2.

The deletion of documents becomes more challenging. More precisely, our cleaning process to remove outdated encrypted identifiers needs further adaptations. Removing those causes gaps in the counter sequence. Table 4 shows the problem. Suppose the documents matching $keyword_2$ (id_2, id_4 and id_5) should be deleted. The last three entries of the index related to $keyword_2$ will be removed immediately. id_2 and id_4, however, remain as entries of $keyword_1$ (highlighted) but can be deleted gradually once $keyword_1$ is queried. If this happens, in order to close the gap between the counter values 0 and 2, the client needs to reorganize (or reindex) the encrypted identifiers for $keyword_1$. The client tells the server to delete all entries from the index related to $keyword_1$. For each remaining document identifier id_1 and id_3, the client creates new obfuscated keywords PRF($keyword_1 \parallel 0$) and PRF($keyword_1 \parallel 1$), and sends them together with the encrypted identifiers Enc(id_1) and Enc(id_3) to the server to insert them into the index.

5.3.1 Discussion

In the previous section we presented an extended version of our basic protocol. The security assumptions about the search and access pattern are the same. However, our extended version offers forward privacy by providing a better way to obfuscate keywords. In contrast to the basic protocol, the extended one fits applications where security is more important. On the other hand, there are also some drawbacks.

The runtime of the extended protocol depends on the counter. Our basic protocol provides optimal performance. No matter how many documents are associated with a certain keyword, it takes one hit to get them all. Using a counter, however, the server must look up $PRF_c = PRF(keyword \| c)$ for $c = 0...i$ entries. We note that the counter is never higher as the amount of stored documents because in the worst case a keyword is associated with all documents. Due to this, our extended protocol provides better performance than approaches that do not use an inverted index.

Note that it is not possible to adapt the counter technique to the OPE scheme. In contrast to PRF, an OPE's ciphertext is related to its corresponding plaintext and hence altering the input does not result in a completely different ciphertext. To preserve the order of OPE ciphertexts, randomness must not be added. In case coordinates (specifying a bounding box) are keywords, forward privacy is not ensured.

Using the server as counter store causes another issue. The counter can only be used in a blocking way. During an add operation, nobody else should have access to this value. Otherwise, if two add operations are running concurrently by the same user but different

```
Key k_OPE, k_AES, k_PRF
Collection indexEntries
Collection documents

Map counters = getCountersFromServer(keywords)

for each document in file, do:
    documentID = Identifier()

    // encrypt document
    encDocument = Enc(k_AES, document)
    encDocument.attach(documentID)
    documents.add(encDocument)

    // index security
    for each keyword, do:
        if keyword is of type bounding box
            obfuscatedKeyword = (OPE(k_OPE, minX), OPE(k_OPE, minY), OPE(k_OPE, maxX),
                OPE(k_OPE, maxY))
        else
            counter = counters.getOrDefault(keyword, 0)
            obfuscatedKeyword = PRF(k_PRF, keyword || counter)
            counters.put(keyword, ++counter)

        encryptedID = Enc(k_AES, documentID)
        indexEntries.append(obfuscatedKeyword, encryptedID)

indexEntries.appendAll(encryptCounters(counters))
indexEntries.sort()
sendToCloud(documents)
sendToCloud(indexEntries)
```

Listing 2 Extended single keyword search: add operation with counter. Changes to our basic protocol are underlined.

clients, the counter cannot be incremented correctly. Using a counter value more than once would subvert our security and undo the improvements the counter has achieved in the first place. Our entire add operation is therefore blocking. This may be acceptable in a system like ours where only the data owner can add documents to the store. It is unlikely that the data owner adds two documents from different devices simultaneously. In a multi-writer system, on the other hand, the usability will suffer from the fact that only one client is able to add new documents at a time. This drawback is acceptable for now, because our protocol does not support a multi-writer feature. But in case we introduce such a feature in the long term, this issue must be considered.

5.4 Boolean Expressions

Besides single keyword searches our system should also support more complex queries involving multiple keywords combined by boolean operators (i.e. AND, OR and NOT). A naïve technique for this task is to perform the computation on the client side instead of the server side. For the AND operator, the client runs one search per keyword independently and receives a collection of identifiers for each query. The client then calculates the intersection of all collections.

NOT queries can be performed similarly. The client sends the keyword to negate and the server replies with two collections. One includes the identifiers of the keyword's search outcome, the other all identifiers from the index. Again, the client is now responsible to filter these results by discarding the intersection of both collections. The OR operation is straight forward because we perform two keyword searches independently and combine their results in the end.

This technique has two drawbacks in terms of performance and security of SSE. The server sends a lot more encrypted identifiers than necessary to perform the query. Performing any NOT operation, the server must send the entire index, so the client is able to identify relevant results. Bandwidth is therefore wasted. The security concern is even more critical and caused by the fact that the server learns the complete outcome of each keyword within a boolean expression. Especially the AND operation is leaking more information than necessary if a client searches for two high-frequency keywords whose conjunction applies to only a small number of documents. The relation between the result of a

queried keyword and the final outcome allows the server to draw conclusions about the query.

A better approach to support boolean expressions in SSE was published by Cash et al. [1]. Their protocol is called OXT and provides an effective, yet secure technique to handle logical operators in SSE queries. The server does not learn anything about the result of a single keyword within a query. We embed the approach of Cash et al. into our single keyword protocol from the previous section and adapt it to our requirements and assumptions.

5.4.1 BXT and OXT

Cash et al. introduce OXT by first presenting an easier protocol called Basic Cross-Tags (BXT). BXT provides the same functionality as OXT but leaks a little more information. Pointing out the leakage's impact, Cash et al. leave it to the reading developers to decide which protocol is more suitable for them. In this section, we explain their differences and conclude in which scenario one of them is preferable. Both extend single keyword search techniques by boolean expressions.

BXT (and OXT) focuses on one logical operator at a time. An OR operation is the least complex one because the query is simply split and the searches for both keywords are performed independently. AND and NOT however are more complex. To deal with these two operators another piece of information is required. Cash et al. call it *xtag* in their protocol. An xtag is a string which acts as a flag indicating whether a keyword/document association exists or not. Its actual content is not important but its existence. Xtags are generated during the add operation on the client side and are sent to the server besides the encrypted documents and index entries. For each keyword/document association an xtag is created by the client. Its usage however is locked by a secret, called *xtrap*. These values ensure that the server is not able to handle xtags without the client.

Listing 3 illustrates how xtags are generated during an add operation. The client performs a PRF and uses its result (the xtrap) as key to run the PRF once more. The cryptographic key used to generate xtraps must not be the same as the one used to obfuscate keywords. Otherwise, the xtrap would be equal to the corresponding obfuscated value stored in the index. All xtags are sent to the server which stores them besides the index. The xtraps do not leave the client but are dropped.

Once a search is running, the query is transformed into a *Searchable Normal Form (SNF)*, which is specified by the form $w_1 \wedge \Phi(w_2, ..., w_n)$ where $w_1, ..., w_n$ are the keywords to search for and Φ is an arbitrary boolean operator. The first keyword w_1 is very important because the search performance highly depends on it (as we show in Section 7). For the remaining query $\Phi(w_2, ..., w_n)$, the client regenerates xtraps for w_2 to w_n. Xtraps and the regular obfuscation of w_1 are sent to the server. The w_1 keyword is handled similarly to a single keyword search on the server side resulting in a collection of associated document identifiers. The server then checks for each identifier if its combination with all xtraps results in a known xtag or not. Only now is the server able to handle xtags and only the ones belonging to the received xtraps. If the xtrap belongs to an AND expression, all xtags must exist on the server side. In case one is unknown, the identifier does not satisfy the AND expression and can be discarded. NOT expressions work the other way round. If at least one xtag exists on the server the test will fail. All identifiers that pass their corresponding check are part of the final search outcome. Listing 4 shows the xtag check assuming the server has already received the list of document identifiers from the single keyword search for w_1.

```
Key k_BXT
for each document in file, do:
    documentID = Identifier()
    for each keyword, do:
        xtrap = PRF(k_BXT, keyword)
        xtag = PRF(xtrap, documentID)
```

Listing 3 xtag generation for boolean expressions

```
Boolean result = true

for each documentID, do:
    for each xtrap, do:
        xtag = PRF(xtrap, documentID)
        if (xtag exists on server)
            if (Φ is NOT operator)
                // NOT expression is false
                // for this documentID,
                // because no xtag must
                // exist on the server
                result = false
        else
            if (Φ is AND operator)
                // AND expression is false
                // for this documentID,
                // because all xtags must
                // exist on the server
                result = false
```

Listing 4 Boolean expression: xtag check during a search

BXT suffers from a certain leakage motivating Cash et al. to design OXT as enhancement [1]. Once the client exposes xtraps as part of a search, the server can reuse them to test identifiers from previous or following queries. The protocol therefore leaks information

```
Key k_OPE, k_AES, k_PRF, k_BXT

Collection indexEntries
Collection documents
List xtags

for each document in file, do:
    documentID = Identifier()

    // encrypt document
    encDocument = Enc(k_AES, document)
    encDocument.attach(documentID)
    documents.add(encDocument)

    // index security
    for each keyword, do:
        if keyword is of type bounding box
            obfuscatedKeyword = (OPE(k_OPE, minX), OPE(k_OPE, minY), OPE(k_OPE, maxX),
                OPE(k_OPE, maxY))
        else
            obfuscatedKeyword = PRF(k_PRF, keyword)
            xtrap = PRF(k_BXT, keyword)
            xtag = PRF(xtrap, documentID)

        encryptedID = Enc(k_AES, documentID)
        indexEntries.append(obfuscatedKeyword, encryptedID)
        xtags.add(xtag)

indexEntries.sort()
sendToCloud(documents)
sendToCloud(indexEntries)
sendToCloud(xtags)
```

Listing 5 Boolean expression: add operation on the client side. Changes to our basic protocol are underlined

across queries allowing the server to learn intersections between them. OXT extends the protocol by introducing an alternative to how xtraps are handled. Instead of revealing all xtraps during a search, client and server execute a secure two-party computation [18]. This allows two parties to jointly execute a common function over their inputs without leaking those to the other party. Both parties can be sure the final result is correct. Using a secure two-party computation protocol, client and server can jointly generate xtags without forcing the client to leak any information about the xtraps of a query. However, secure two-party computations are expensive and require several communication rounds.

5.4.2 Integration

In comparison to the assumptions of BXT (and OXT), we need an extra step to integrate the protocol in our single keyword search. BXT assumes that the resulted identifiers are in plaintext as soon as the search for w_1 is finished. Regarding our own protocol, they are not and must be decrypted by the client first. In our protocol, this does not cause an extra communication round since the client sends all decrypted identifiers back to the server anyway. Listing 5 shows the add operation of our basic protocol including the xtag generation to support boolean expressions.

The integration of boolean expressions into our protocol has a limitation regarding bounding box queries. BXT and OXT do not support range queries. Xtags only indicate if a keyword/document association exists but do not tell anything about the keyword itself. If a bounding box is part of a boolean expression, it must always be used as w_1 regardless of its frequency. This implies that only one bounding box can be handled at a time because we set only one w_1 keyword. This affects our system's usability only in one case, because our SSE system is primarily optimized for geospatial file storage. We have introduced range capabilities to cover bounding box-related queries. Given that this feature is exclusively used for this task, we note that a conjunction of two bounding boxes does not make sense because if an object is within two bounding boxes, they blend and their intersection can be used instead. A disjunction of bounding boxes is also no issue because in that case we split the query anyway. The only limita-

tion are NOT-related expressions. An example would be if a user specifies a bounding box but wants to exclude a certain section inside (represented by a smaller bounding box). This kind of queries is not supported. Generally, we note that only one range query at a time is supported by BXT and OXT.

Another question is which of our protocols interacts best with BXT and OXT respectively. We have presented two single keyword protocols in the previous sections, a basic version and an extension. Theoretically, we can combine each with each resulting in four protocols. Our basic protocol combined with BXT would achieve the best performance, our extended one combined with OXT the highest security. The leakage of one of our single keyword protocols does not neutralize the security of BXT or OXT and vice versa. This is mainly due to the fact that both operate on different data. BXT and OXT deal only with xtrap and xtag, our single keyword protocols with obfuscated keywords. Therefore, the security assumptions—such as forward and backward security—for the single keyword search are the same as described in the corresponding sections of our basic and extended protocol.

BXT and OXT limit the delete capabilities of our protocols. Xtags, which are once created can not be removed easily on the server side. During a delete operation, the client owns all identifiers referring to documents that should be removed from the store. Nevertheless, the client has no chance to find all corresponding xtags but only the ones from the current query. If the documents have been associated with further keywords, these xtags remain on the server. A technique to delete all xtags efficiently remains as future work.

6 Implementation

In our theoretical part, we have described four protocols. Two of them focus on single keyword searches namely our basic protocol and the extended version with counters. Moreover, we have described two extensions (BXT and OXT) to support boolean expressions on top of the single keyword searches. The basic protocol and BXT as extension for boolean expression support have been implemented as part of GeoRocket.

6.1 Import Command

The import command refers to our add operation in SSE. The main work happens on the client side: We parse the geospatial file and extract relevant keywords such as user-specific tags and bounding boxes. Each keyword is associated with a generated unique identifier (UUID [13]) referring to the chunk in the file the keyword was extracted from. For example, a geospatial file containing a 3D city model is split into chunks representing individual buildings. Each building gets a unique identifier and the keywords such as the building's street name, the house number, etc. are linked to it. The keywords are obfuscated using either PRF or OPE. We use HMAC as PRF algorithm and the OPE implementation from CryptDB [3]. Identifiers are encrypted with AES. Additionally, we generate an xtag using the keyword and UUID of the current chunk (the plaintext identifier). The obfuscated keyword and encrypted identifier are stored in a map representing our SSE index. Afterwards, the chunk itself is encrypted symmetrically using AES, and the UUID is attached as part of the encryption header. After all elements have been traversed and all keywords have been found, we shuffle the list of xtags and the SSE index entries.

Figure 4 shows an overview of the workflow between client and the server. The client sends the set of encrypted chunks and the SSE index ❶ to the GeoRocket server. The server stores the encrypted chunks in the configured storage backend (e.g. local file system or Amazon S3) ❷ and adds the SSE index to Elasticsearch ❸.

6.2 Search Command

To run a search, users define a query consisting of the keywords they are looking for. Our support of boolean expressions includes AND and NOT operators. Nested queries such as NOT(AND(k1 k2)) are not supported because their compilation into a usable formula is beyond the scope of this paper. The client selects a non-negated

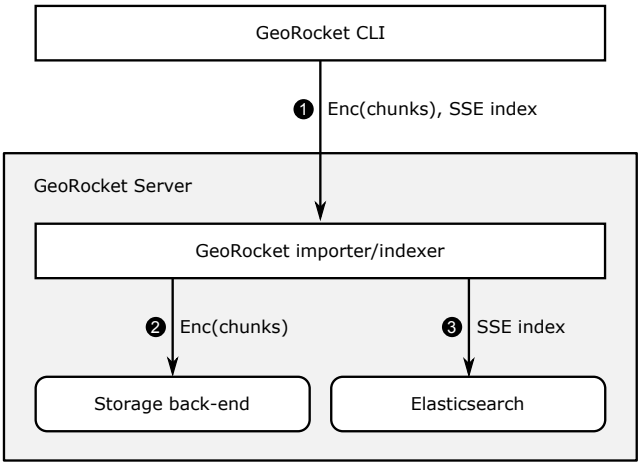

Fig. 4 Workflow of an import command in SSE

keyword of the query and marks it as the least frequent keyword. We pick the first one, which can be a bounding box. This special keyword is obfuscated either with OPE or PRF. The remaining keywords are replaced by their corresponding xtraps. Of course, the logical operators stay the same. This information is encoded into a request and sent to the server. The interaction process is shown in Figure 5.

The server performs a single search on the least frequent keyword ❶. Doing so, the SSE index is searched by performing an Elasticsearch query. Its outcome is a list of Elasticsearch documents that include the obfuscated keyword or—in case it was a range query—whose bounding box matches the search criteria. The server extracts the encrypted identifiers and sends them back to the client ❷ where they are decrypted and returned with the remaining query (including the xtraps) ❸. We avoid the complex secure two-party computation of OXT as we have implemented the BXT protocol to handle xtraps. The collection of identifiers is tested against the remaining query. If one generated xtag exists in Elasticsearch and the xtrap belongs to a NOT operator, the identifier is dropped. If one generated xtag does not exist and we deal with an AND operator, the identifier is also dropped. Filtering the collection, the server obtains the final search results: a collection of identifiers referring to chunks that satisfy the query. Since these identifiers are the UUIDs of chunks and have been indexed during the import process, the server is able to select them from its file store. The chunks are merged and the result is sent back to the client which decrypts it as last step ❹.

6.3 Delete Command

Our delete command is similar to a search. Instead of returning the matching chunks in the last step, the server deletes them. Additionally, all Elasticsearch documents containing the keyword are removed to clean the SSE index. As mentioned before, xtags cannot be completely deleted from the server because an efficient deletion technique for BXT is beyond the scope of this paper.

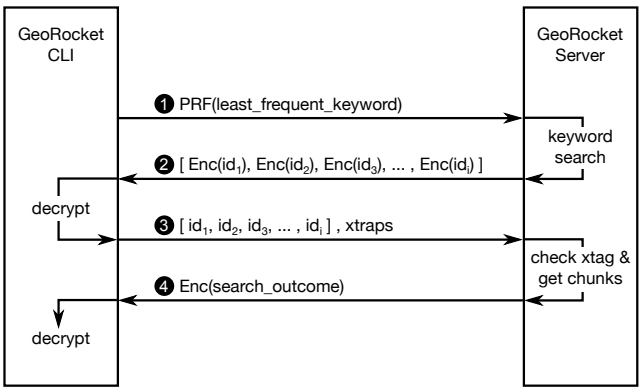

Fig. 5 Workflow of a search command in SSE

7 Evaluation

In this section, we evaluate our protocol based on performance measurements. For this, we implemented our basic protocol with BXT. Test results for the other protocols are not included in this paper.

We compare the runtime of our import and search operations regarding encrypted (SSE) and unencrypted data (non-SSE). We do not evaluate the delete operation. Deleting a single keyword works similarly to the search operation. Additionally, as described above, the delete operation of BXT and OXT is beyond the scope of this paper. All performance measurements were executed 100 times to avoid getting skewed results due to fluctuations, and we calculated the mean values accordingly. We performed the tests with different server setups. The client, however, ran always on a machine with the following system specifications:

- CPU: Intel® Core™ i5-6600T @ 2.70GHz × 4
- Memory: 8 GB (2 × 4096 MB)
- OS: Linux, Ubuntu 16.04 LTS 64-bit
- Java Runtime: openjdk-8-jdk (1.8.0_121)

In Section 7.1 we present our results from running the server on the same machine as the client. In addition, we set up a number of virtual machines in a cloud where we deployed the GeoRocket Server, a distributed MongoDB database storing the chunks as well as a distributed Elasticsearch index. The results are presented in Section 7.2.

7.1 Single Machine

The following tests were performed on a machine running client and server simultaneously. The system specifications are shown above.

7.1.1 Import

Using our SSE import, the client parses the file and locates its chunks which are then indexed according to our protocol. The client obfuscates the keywords via an OPE or PRF transformation and encrypts the chunks. The counterpart operation (non-SSE) sends the unencrypted file directly as no more steps are necessary on the client side. We tested several files which are shown

in Table 6. All of them are real-world datasets from the open 3D city model of Rotterdam [15] without further modification. The files differ in their sizes and accordingly have different amounts of chunks (i.e. buildings), regular keywords and bounding boxes. For SSE, the import highly depends on the file, especially on its number of keywords (including bounding boxes). The more keywords a file contains the more times our protocol has to run the obfuscation process.

Size	#Chunks	#Keywords	#Bounding Boxes
2 MB	97	4 968	97
5 MB	390	15 125	390
7 MB	426	21 138	426
15 MB	751	44 676	751
37 MB	2 324	112 748	2 324
66 MB	4 741	198 128	4 741
94 MB	6 807	283 528	6 807

Table 6 Varying files for our tests

On the server side, the SSE import does not require much effort. The server receives encrypted chunks and SSE index entries. Handling the imported file is not different to the non-SSE operation because chunks are also stored individually and then processed by the regular index procedure. Using SSE, this indexing is limited to meta information and the chunk identifier because the server cannot read the encrypted content. The only additional step in our SSE system is storing the received SSE index entries in Elasticsearch. This has a slight impact on performance.

Figure 6 shows the test results. The chart shows that the import time increases in relation to the file size for both SSE and non-SSE. This was to be expected. The larger a file is the longer it takes to process it. Using the non-SSE import, the time increases only a little over growing sizes. The performance is very good even for larger files, because the client transmits the file to a server running on the same machine. To import a 2 MB file, only 0.08 seconds are required. 94 MB are imported within 0.58 seconds. Our import in SSE on the other hand requires more runtime. For the smallest file, the operation takes 18 times longer (1.51 seconds) and for the largest file 33 times (19.53 seconds). On average, the SSE import takes 22 times longer than importing files in the non-SSE system.

7.1.2 Search

More important than the import time is the search time. A file is usually imported once but searched many times. Testing the search performance is also complex because this procedure highly depends on the kind of queries, as well as how many chunks and keywords are stored on the server. We perform these tests by preparing two stores and then measure the search time for different queries. While one store implements SSE, the other performs the searches without it. Both stores are strictly separated but include the same files (in case of SSE they are encrypted). Each store has a total size of 321 MB and includes 15 536 chunks.

For each chunk we store one document in Elasticsearch containing the regular meta information. The non-SSE store does not require any other information because the chunk documents include all indexed keywords.

The SSE store maintains keywords and bounding box-related documents separately as part of the SSE index. In addition to that, we generate an xtag for every keyword/document association unless it belongs to a bounding box. As a consequence, the number of xtags and keyword documents are equal. Overall, the store maintains 1 452 908 documents (see Table 7).

Meta Information	Count
# Documents (Chunk)	15 536
# Documents (BB)	15 536
# Documents (Keyword)	710 918
# Documents (Xtag)	710 918
# Documents (Total)	1 452 908

Table 7 Meta information of the SSE store

Our first test covers single keyword searches. Seven files have been introduced in the beginning of this section (see Table 6), all with different amounts of keywords to index. We add some user-specific tags to each of them and run a test for each file respectively. For this test, we measure the total search time, starting when the command is executed on the client side until the entire search outcome has been rendered to the client (including the decryption in case of SSE). Figure 7 shows the results.

The runtime increases with growing search results for both operations. Merging and transmitting the result to the client takes longer the bigger the result is. Again, the SSE search requires more time. If the search result is of size 5 MB, the total time is 0.14 and 0.79 seconds for our non-SSE and SSE operation respectively. On average, the SSE search takes nine times longer than searching in a non-SSE store. The difference is therefore not as high as in our import operation.

The runtime of bounding box queries is very similar to the single keyword search. Instead of performing a PRF, the client uses OPE to transform the coordinates of the bounding box. This generation is more

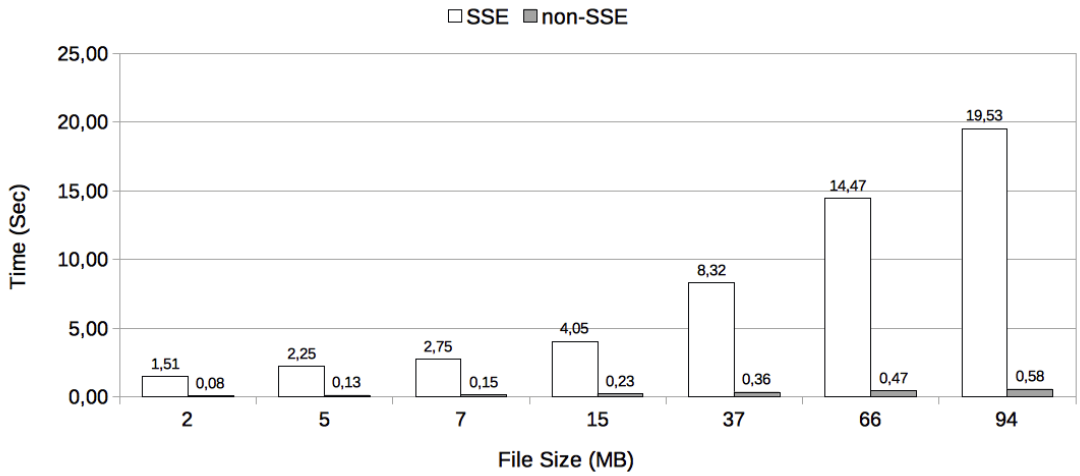

Fig. 6 Import on the client side: SSE vs. non-SSE. Client and Server are running on the same machine.

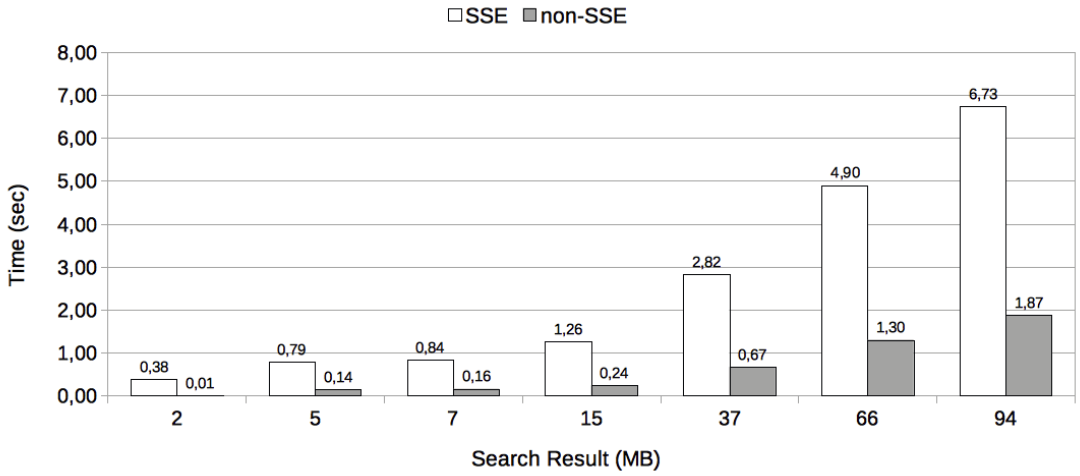

Fig. 7 Single keyword search: SSE vs. non-SSE. Client and Server are running on the same machine.

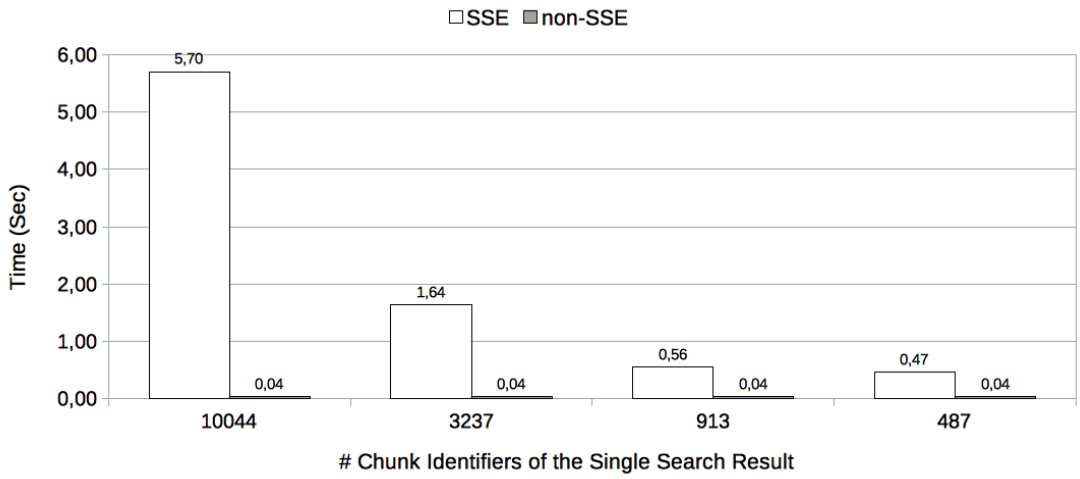

Fig. 8 Boolean expression search: SSE vs. non-SSE. Client and Server are running on the same machine.

time-consuming than running a PRF but since we can only search for one bounding box at a time, the client only performs four OPE transformations—one for each coordinate. The search time increases negligibly. The server is able to compare several OPE ciphertexts as fast as plaintext ones. The final decryption is equal to our regular single keyword search and thus equally fast.

As last step, we measure the runtime of a search with boolean expressions. For this test, we use the same store on the server side as before but create a more complex query. The client's tasks are as follows: with regards to SSE, the query is compiled according to our protocol by marking the first keyword as the least frequent one and replacing all remaining keywords with their corresponding xtraps. The server performs a search on the marked keyword and finally returns all encrypted chunks whose generated xtags are either all known to the database (AND expression) or not known at all (NOT expression). Using a non-SSE search on the other hand, the client sends the query and the server selects the respective chunks and replies with the merged geospatial file in plaintext. Our query consists of 5 logical operands: one is negated (NOT) and the others are joined using AND. The returned geospatial file includes 97 chunks. Every keyword within the query is associated with a different amount of chunks, which is shown in Table 8. For this test, we run the search four times using the queries presented in Table 9.

Keyword	# Chunks
alpha	10 044
beta	3 237
gamma	913
delta	487
epsilon	390

Table 8 Keywords and their chunks

Our queries differ as each one starts with another keyword. This keyword is always selected as least frequent one and therefore is queried as part of a single search. To execute our first query, the server runs a search on *alpha* resulting in 10 044 chunk identifiers. After the client has decrypted them, the server generates the xtags of all those identifiers in combination with the remaining query and finally checks if the xtags are known to the database or not. Our second query results in 3 237 identifiers to generate xtags from. The other queries are created analogically. The keyword *epsilon* is never selected as it belongs to a NOT expression. By swapping the first keyword, our xtag generation starts with a different amount of identifiers each time. The final search result however is the same for all four queries.

	Query with Boolean Expression
1	AND(alpha beta gamma NOT(epsilon) delta)
2	AND(beta alpha gamma NOT(epsilon) delta)
3	AND(gamma beta alpha NOT(epsilon) delta)
4	AND(delta beta gamma NOT(epsilon) alpha)

Table 9 Queries with boolean expressions

We evaluate how this impacts the total search time. Figure 8 shows the results.

The chart shows that the search time highly depends on the keyword which is selected as least frequent. If 10 044 identifiers must be checked for their corresponding xtags, the search requires 5.70 seconds. It decreases with the amount of identifiers which are returned as part of the single search result. The best performance is achieved if the first keyword is linked to as few chunk identifiers as possible (0.47 seconds). By selecting the keyword wisely, we can save 92% of the runtime. In comparison to that, the non-SSE search does not depend on the query's order. The runtime is always the same (0.04 seconds). Nevertheless, the SSE search still requires more runtime than its counterpart.

Not all aspects have been included in these tests. Our SSE search protocol requires one more communication round since the client has to decrypt the chunk identifiers. We do not measure latency in our tests, because we run all of them on the same machine. This section shows that operations on encrypted data require more runtime in comparison to plaintext data, although we have implemented the basic protocol with BXT which should achieve the fastest results of all our encryption protocols. However, the runtimes are promising to achieve an acceptable performance for real-world applications.

7.2 Cloud

We tested the performance of searching in encrypted data in the cloud. For this experiment we set up 7 homogeneous virtual machines with the following specifications:

- CPU: 8 cores
- Memory: 8 GB
- OS: Linux, Ubuntu 16.04 LTS 64-bit

Figure 9 depicts our setup. We installed Elasticsearch on three virtual machines and configured it to use sharding. We also installed MongoDB on three other virtual machines and configured it for replication. We used the last virtual machine for GeoRocket.

The cloud we used to host the virtual machines is based on OpenStack and has the following specifications:

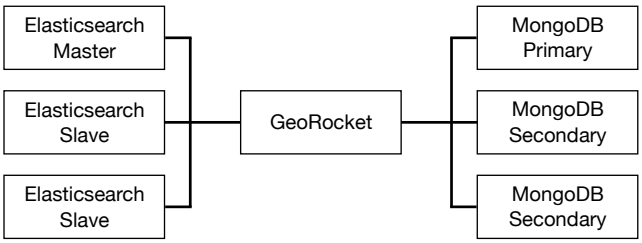

Fig. 9 Cloud setup for our test

- Controller nodes: 3
- Compute nodes: 13
- Storage nodes: 6
- Total numbers:
  - CPU: 448 cores
  - Memory: 4 TiB
  - Storage: 250 TiB

All nodes were connected via 2 × 10 Gbit/s Ethernet. The client and the virtual machines in the cloud were communicating over 100 Mbit/s Ethernet. The client's system specifications were the same as in our previous tests.

We did not test the performance of the import process because the relevant parts of it run on the client side. Therefore, the results from Section 7.1 apply here too. We also did not test boolean expressions in the cloud. This test was performed to show the importance of the least frequent keyword and did not depend on the server. We therefore executed the tests regarding single keyword searches only. The test results are presented in Figure 10.

The runtime increased in comparison to the tests we had executed on a machine running client and server simultaneously (see Figure 7). Because the GeoRocket Server and the CLI are deployed on different machines, the data transmission between them takes longer. This affects both our SSE and non-SSE operation. If the search result is 94 MB large, the total time is 10.42 seconds for our non-SSE operation in comparison to 1.87 seconds we achieved on a single machine. This is as expected considering a maximum connection rate of 100 Mbit/s (94 MB ÷ max. 100 Mbit/s = at least 7.5s).

The total time for SSE is 0.47 seconds if the search result is 2 MB large and 12.49 seconds for 94 MB. Comparing the non-SSE and SSE operation in the cloud, the search times are much closer now. Although the SSE operation also depends on the connection rate, its search time did not increase as much as the time of its non-SSE counterpart. This is mainly due to the fact that Elasticsearch provides several optimizations when it is deployed in a distributed manner allowing for an increased performance. The index is divided and distributed across multiple nodes and queries can be parallelized.

Our SSE operations benefit more from these optimizations than the non-SSE ones, because our SSE system interacts more often with Elasticsearch. The SSE index is entirely maintained by Elasticsearch, searching in encrypted data therefore highly depends on the search engine. The non-SSE operations do not require as many Elasticsearch requests.

All in all, the transmission time must be added, but the search operation itself is faster in the cloud, which becomes particularly evident in our SSE system.

8 Conclusions

In this paper, we designed, implemented, and evaluated an SSE system for geospatial file storage. We focussed on an approach to encrypt an inverted index and to protected it against an honest-but-curious server while maintaining the possibility to query the index by a client. Our file storage is dynamic as we allow users to add more documents after the index has been initialized. We introduced multiple protocols that are suitable for the index part. They differ in their respective security level or rather in the amount of their leakage. Our research shows that the amount of leakage highly depends on the performance we want to achieve. The features of the protocols differ too. Our single keyword search supports bounding box-related queries. In addition, it allows for deletion of documents. However, BXT and OXT limit this feature because documents can be deleted but xtags remain on the server.

We described two protocols to perform a single keyword search. One achieves a very good performance because the server only needs one hit to get all documents. The performance of our extended protocol relies on a counter that is, in the worst case, equal to the amount of stored documents. Cash et al. published two protocols extending the single keyword search by two logical operators (AND and NOT). They differ in their leakage level and speed respectively. We discussed the leakage's impact and which protocol is acceptable in different scenarios. We implemented the basic protocol with BXT as an extension to GeoRocket.

To ensure confidentiality, the actual geospatial files, or rather the chunks extracted from them, are encrypted symmetrically using AES. The leakage is small because an honest-but-curious server can only learn the chunk size and how many chunks have been extracted from the imported file. We evaluated our entire SSE system and compared it to the regular import and search process (on plaintext data). Both operations take much longer

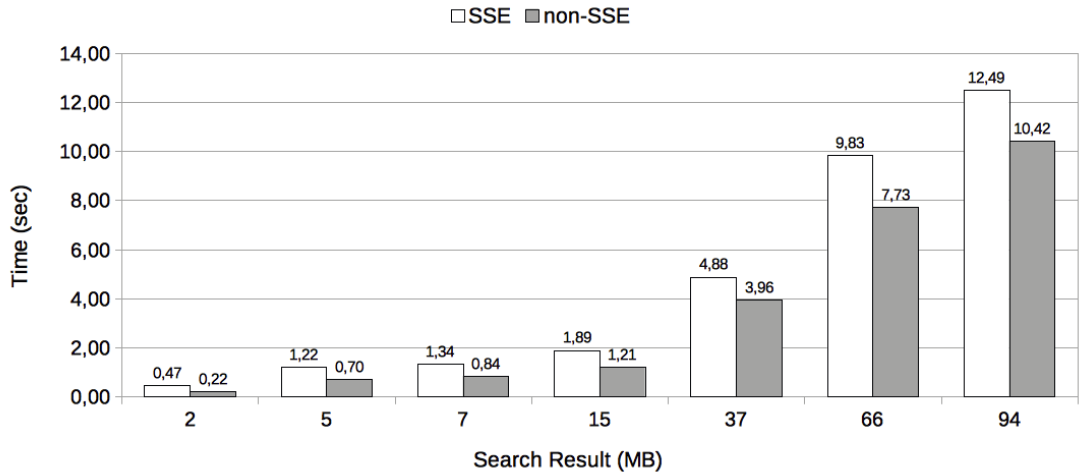

Fig. 10 Single keyword search: SSE vs. non-SSE. The server is deployed in the cloud.

in SSE. We also performed our evaluation in a cloud. The search time decreases for SSE in such an environment (without regard to the transmission time) because the operations benefit from the distributed Elasticsearch index.

In summary, SSE fits well in geospatial file storage. We enable users to send their sensitive data to the cloud without having to worry that the content's confidentiality is harmed. SSE supports a keyword-oriented search in encrypted data while hiding the queried keywords. With our approach users can handle their confidential data in GeoRocket, but our SSE system can also be integrated into other geospatial storage solutions.

Although GeoRocket targets geospatial data storage, it can actually handle arbitrary structured files. This enables further applications as our keyword-based search does not depend on geospatial features. Our approach can therefore be applied to other areas dealing with keyword-based searches in encrypted data in general.

Of course, security can never be fully ensured. We discussed several leakages in the course of this paper which allow an honest-but-curious server to learn some information about queries or the stored files. Besides that, security is not a state but a process requiring both users and developers to stay informed about new threats and handle their data in a responsible way with regards to its importance and the required confidentiality. However, our SSE system shows that users do not need to avoid the cloud but can benefit from its advantages and protect their data at the same time.

8.1 Future Work

In this last section, we introduce some work that can be done to continue our research. Although SSE has made a huge progress in the last decade, some issues are still open. Boolean expressions need further improvements like the support for nested expressions. This is not exclusively a security-related issue because the easiest way to handle nested expressions is by fragmenting them. The transformation of boolean expressions while preserving their truth table is useful in many areas. In addition, a strategy to find the least frequent keyword within a query is important because the performance depends on it. An generic strategy regarding our requirements remains as future work.

Regarding OXT and BXT, future work is needed to improve the deletion of xtags. During a delete operation, neither client nor server are able to identify all xtags for a certain document. Over time this causes growing storage.

Our protocols assume an honest-but-curious server, that answers all requests strictly to the protocol. The client is not able to verify the correctness of the server's responses—e.g. if all returned encrypted identifiers really belong to the client's request. Future work should address the threat of a malicious server and how clients could be protected from malicious responses.

More search features should also be supported in addition to our range queries. Examples for other features are wildcard and substring searches. Some papers already describe the theoretical part of these features in an SSE environment [6], but more work with a focus on real-world applications is necessary in this field. SSE should support functionalities similar to the ones in plaintext search engines.

We do not support multi-party settings so far. Our protocol involves a single data owner who has exclusive read and write access to the encrypted data. Multi reader or writer settings are useful in company projects in which members share geospatial data in a group. These settings should be addressed in future work.

Acknowledgements We would like to thank Ralf Gutbell for his thorough review and the valuable input.

References

1. Cash D, Jarecki S, Jutla C, Krawczyk H, Roşu MC, Steiner M (2013) Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries, Springer, Berlin, Heidelberg, pp 353–373. DOI 10.1007/978-3-642-40041-4_20
2. Cash D, Jaeger J, Jarecki S, Jutla CS, Krawczyk H, Roşu MC, Steiner M (2014) Dynamic searchable encryption in very-large databases: Data structures and implementation. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014
3. CryptDB (2015) `http://css.csail.mit.edu/cryptdb/`, [Online; accessed 30-Jan-2017]
4. Elasticsearch (2017) `https://www.elastic.co/products/elasticsearch`, [Online; accessed 23-Jan-2017]
5. European Union (2016) Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) - OJ L 119 (EN), p 8. Article 45
6. Faber S, Jarecki S, Krawczyk H, Nguyen Q, Rosu M, Steiner M (2015) Rich Queries on Encrypted Data: Beyond Exact Matches, Springer International Publishing, pp 123–145. DOI 10.1007/978-3-319-24177-7_7
7. Gentry C (2009) A fully homomorphic encryption scheme. PhD thesis, Stanford University
8. GeoRocket Website (2017) `http://georocket.io`, [Online; accessed 10-Jan-2017]
9. Goldreich O (1987) Towards a theory of software protection and simulation by oblivious RAMs. In: Proceedings of the 19th ACM Symposium on Theory of Computing, ACM, New York, NY, USA, STOC '87, pp 182–194, DOI 10.1145/28395.28416
10. Hahn F, Kerschbaum F (2014) Searchable encryption with secure and efficient updates. In: Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security, ACM, New York, NY, USA, CCS '14, pp 310–320, DOI 10.1145/2660267.2660297
11. Kamara S, Papamanthou C (2013) Parallel and Dynamic Searchable Symmetric Encryption, Springer, Berlin, Heidelberg, pp 258–274. DOI 10.1007/978-3-642-39884-1_22
12. Kamara S, Papamanthou C, Roeder T (2012) Dynamic searchable symmetric encryption. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, ACM, New York, NY, USA, CCS '12, pp 965–976, DOI 10.1145/2382196.2382298
13. Leach P, Mealling M, Salz R (2005) RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace
14. Ostrovsky R (1992) Software protection and simulation on oblivious RAMs. PhD thesis, Massachusetts Institute of Technology (MIT)
15. Rotterdam Open Data Store - Rotterdam 3D (2014) `http://rotterdamopendata.nl/dataset/rotterdam-3d-bestanden`, [Online; accessed 10-Jan-2017]
16. Song DX, Wagner D, Perrig A (2000) Practical techniques for searches on encrypted data. In: Proceedings of the 2000 IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, SP '00, pp 44–55
17. Stefanov E, Papamanthou C, Shi E (2014) Practical dynamic searchable encryption with small leakage. In: Network and Distributed System Security (NDSS) Symposium, vol 71, pp 72–75
18. Yao AC (1982) Protocols for secure computations. In: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, USA, SFCS '82, pp 160–164, DOI 10.1109/SFCS.1982.88
19. Yavuz AA, Guajardo J (2015) Dynamic Searchable Symmetric Encryption with Minimal Leakage and Efficient Updates on Commodity Hardware, Springer International Publishing, pp 241–259. DOI 10.1007/978-3-319-31301-6_15