View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Stirling Online Research Repository

Towards Search on Encrypted Graph Data

A. Kasten[1], A. Scherp[2], F. Armknecht[3], and M. Krause[3]

[1] University of Koblenz-Landau, IT Risk Management, Germany
{andreas.kasten}@uni-koblenz.de,
[2] University of Mannheim, School of Business Informatics & Mathematics, Germany
{ansgar}@informatik.uni-mannheim.de
[3] University of Mannheim, Theoretical Computer Science & IT Security, Germany
{armknecht,krause}@informatik.uni-mannheim.de

Abstract. We present an approach where one can execute user defined SPARQL queries on encrypted graph data. The graph data is only partially revealed to those users authorized for executing a query. The approach is based on eight different types of queries, corresponding to the different binding possibilities in a single SPARQL triple pattern. The allowed queries can be further restricted by the owner of the graph data, e. g., through pre-defining a specific predicate or object. Single triple patterns can be combined to query group patterns as they can be stated in SPARQL queries and allow to execute a wide range of SELECT and ASK queries.

1 Introduction

When one does not want to provide full public access to some graph data, access control mechanisms can be applied like [1] that limit access to the data based on, e. g., roles in an organization or credentials issued to individuals. However, there are scenarios where one even wants to go one step further and encrypt the graph data. For example, the knowledge base might be hosted at a service provider that is not fully trusted such as in the cloud [2]. Legal issues like the Health Insurance Portability and Accountability Act (HIPAA)[4] in the US require to encrypt semi-structured data such as medical records and other sensitive health information [3]. In addition, different parties are allowed to view only different parts of the data. In another scenario, the graph data might be published in a distributed fashion such as in a peer-to-peer network [4]. The data owner does not know which peer keeps a local copy of the data, but he wants to restrict access to specific parties in the network.

A standard approach would encrypt the entire document containing the information to be protected. The only possible operation is to decrypt the entire document, given that the user is in possession of a corresponding decryption key. However, this would not be suitable to the scenarios above as they require a more fine grained and flexible solution to access the encrypted graph information. In

[4] http://www.gpo.gov/fdsys/pkg/CRPT-104hrpt736/pdf/CRPT-104hrpt736.pdf, last accessed: 19.3.2013

recent time, novel approaches such as functional encryption [5] and searchable encryption [6] have been developed to address this issue. Functional encryption defines a (limited) set of functions that can be applied on the encrypted data by different parties. Searchable encryption extends functional encryption by allowing for executing user-defined operations on the encrypted data. Only at run-time, the operands are bound to their concrete values, which allows for implementing typical information system scenarios like those sketched above.

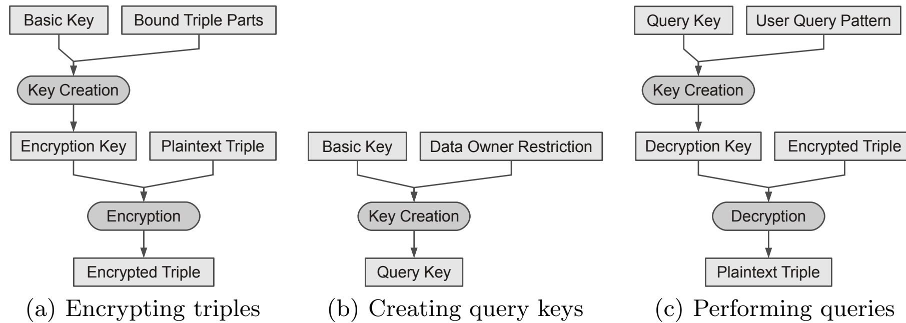
Our approach for search on encrypted graph data makes use of two-layers: The basic layer provides eight encryption keys for eight views on the encrypted data. These eight different views correspond to eight different binding possibilities of a single triple pattern in a SPARQL query. The basic layer can be refined by the data owner on the refinement layer. Here, one can define which kind of SPARQL query patterns can be executed on the encrypted data. For example, the data owner can specify that the users can only request instances of a specific class such as `?x rdf:type foaf:Person`. Single query patterns can be combined to query group patterns to formulate a wide range of SELECT and ASK queries on the encrypted data. Please note, blank nodes are supported in our graph search approach as they are encrypted like resources.

In the subsequent section, we describe in more detail our idea of a two-layered approach for searching on encrypted graph data. The related work is presented in Section 3. In Section 4, we present the basic notations for our approach. In Sections 5 and 6, we present the encryption and preparation of the graph data and the execution of queries on that data.

2 Scenario and Envisioned Functionality

A Semantic Web *document* [7] consists of several triples and is owned by a *data owner*. The data owner encrypts the document and publishes it on the web. Different *users* can access parts of the document by performing *queries* on the encrypted document, i. e., they can search on the encrypted graph document. The data owner defines which users are allowed to perform which queries. This is done by handing over *query keys* to the users through a secure channel. A user performs queries on the encrypted document by using the received query keys and self-defined bindings of the *query patterns*. The result of a query consists of all triples of the document which satisfy the query.

Thus, our work aims at executing queries based on SPARQL [8] triple patterns on the encrypted graph data. These patterns allow each part of a triple to be either bound or unbound. Bound parts correspond to parameters that are specified in the query whereas unbound parts determine the results of a query. Considering all possible combinations of bound and unbound parts in a SPARQL triple pattern results in eight different binding combinations. In our approach, each triple is encrypted individually for each of these eight binding combinations using a corresponding *encryption key*. As depicted in Fig. 1(a), an encryption key comprises the bound parts of a triple as well as a *basic key* which defines the binding combination.



(a) Encrypting triples (b) Creating query keys (c) Performing queries

Fig. 1. Different keys and operations required for searching on encrypted documents.

In order to further restrict the queries that a user can perform on an encrypted document, the data owner defines *restriction patterns*. A restriction pattern narrows down the possible query patterns that the user can define for querying on the encrypted graph data. To this end, the data owner pre-defines in a restriction pattern the bound parts of a SPARQL triple pattern. As depicted in Fig. 1(b), a restriction pattern and a basic key are combined into a query key, which is handed over to the users via a secure channel. The basic key defines the binding combination of a query and the restriction pattern contains the parameters of this query already bound by the data owner.

Performing a query on an encrypted document basically corresponds to decrypting the encrypted triples of this document with a *decryption key*. As depicted in Fig. 1(c), the decryption key is created from the query key received from the data owner and the user's query pattern. It corresponds to a particular query and encodes all bound query parameters taken either from the query key or from the user's query pattern. If an encrypted triple can successfully be decrypted using a decryption key, the triple satisfies the query. If the decryption fails, the triple does not fulfill the query. The decryption of an encrypted triple is successful iff the decryption key is identical to the encryption key. This is the case if the combination of the data owner's restriction and the user's query pattern correspond to the bound triple parts of the encryption key.

3 Related Work

A *searchable encryption* scheme [6,9,10] allows legitimate users to search encrypted documents for specific data entries such as triples. The encrypted document is associated with additional metadata and stored on a (potentially untrusted) server such as the cloud. Users search the encrypted document by executing an interactive search protocol with the server. Thus, they communicate with the data owner in order to execute queries. The users receive all triples (in encrypted form) which comply to their query while the server owner learns as little as possible. There are several fundamental differences between searchable

encryption and our approach: i) Searchable encryption deploys interactive protocols with the server owner while we aim for an offline solution. ii) Searchable encryption usually requires to specify the queries before encrypting the triples. Thus, the encrypted data scales with the number of triples *and* the list of queries while we aim for a solution which supports any possible query. Our solution even allows to include further encrypted triples to the encrypted document at later points in time. iii) Searchable encryption does not support any fine-grained access control. More precisely, while it may be possible that different users can search for different triples, it does not support different query types. In contrast, our goal is that a user making a query q can only figure out all triples that comply to q while learning nothing about the other encrypted triples.

Attribute-based encryption such as [11] allows for a more fine-grained access than traditional encryption schemes. Instead of using only one decryption key for each encryption, the encrypted data can be decrypted with different decryption keys where each key is associated with some attributes. The data can only be decrypted if the decryption key matches the respective attributes. All of the attribute-based encryption schemes proposed in recent years are covered by *functional encryption* (e. g., see [12,5,13]). A functional encryption scheme supports a set of functions and encrypts data such that it can be decrypted for each function using a corresponding decryption key. While functional encryption provides fine-grained offline access control to encrypted data, it does not support searching on encrypted data. In particular, our requirement that users may freely specify parts of their queries is not covered.

Poh et al. [14] as well as Kamara and Wei [15] support *searching on encrypted graphs*. Both approaches are based on pre-computed indices for each supported query. In order to use them for graph-based documents, one would create an index for each occurring object value since it might later be used for a query. Overall, this would result in as many indices as there are different object values. Thus, an attacker would learn how many different object values the graph contains. Our approach does not have this problem since the data is encrypted independently from any particular query. Furthermore, query indices created at encryption time must be re-computed and updated if new triples are added. Our approach allows for adding new encrypted triples without changing the already existing encryptions. Relational database solutions such as CryptDB [16] rely on an online solution with a more or less trusted server. Our approach is designed for offline use. Furthermore, join operations in CryptDB reveal too much information about the encrypted graph as the ciphertexts in the joins are compared bitwise for equality, thus they become countable. Overall, one might obtain an isomorphic graph that reveals the nature of the original graph. XML based solutions such as [17,18] require the user to know the internal structure of the queried XML document. In our approach, no information about the encrypted graph is needed. Performing a query only requires a query key.

4 Basic Notations and Formalization

This section formally defines the basic components of our approach as introduced in Section 2. Their use is described in Sections 5 and 6.

Plaintext Triples and Plaintext Documents The set of all plaintext triples t is defined as $\mathbb{U} \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L})$ with $\mathbb{U}$ as the set of all URIs and $\mathbb{L}$ being the set of all literals. It is $t = (s, p, o)$ with $s \in \mathbb{U}$ being the subject of the triple, $p \in \mathbb{U}$ being the predicate, and $o \in \mathbb{U} \cup \mathbb{L}$ being the object. A graph-based plaintext document d is a set of m triples t_i with $i = 1 \ldots m$. It is $d = \{t_1, t_2, \ldots, t_m\}$.

Encrypted Triples and Encrypted Documents Encrypting a plaintext triple t results in an encrypted triple c. An encrypted triple is a tuple of eight bit strings. Each bit string supports one of the eight different binding combinations of a SPARQL [8] triple pattern as described in Section 2. It is $c = (c_{\texttt{---}}, c_{\texttt{+--}}, c_{\texttt{-+-}}, c_{\texttt{--+}}, c_{\texttt{++-}}, c_{\texttt{+-+}}, c_{\texttt{-++}}, c_{\texttt{+++}})$. The indices `+` and `-` correspond to the existence and non-existence of a binding in the query at subject, predicate, and object position, respectively. For example, $c_{\texttt{-+-}}$ supports SPARQL queries with a bound predicate that retrieve tuples of subjects and objects for each matching triple. Encrypting a plaintext document d results in an encrypted document $d_C = \{c_1, c_2, \ldots, c_m\}$.

Basic Keys A basic key k_b is a bit string of length l and used for encrypting the triples t of a plaintext document d for a particular binding combination. The data owner choses eight different basic keys which are identified as $k_{\texttt{---}}$, $k_{\texttt{+--}}$, $k_{\texttt{-+-}}$, $k_{\texttt{--+}}$, $k_{\texttt{++-}}$, $k_{\texttt{+-+}}$, $k_{\texttt{-++}}$, and $k_{\texttt{+++}}$. Each of these keys is used for creating a particular bit string of the encrypted triples c. For example, the basic key $k_{\texttt{+--}}$ is used for creating the bit strings $c_{\texttt{+--}}$.

Queries, Query Patterns, and Query Keys A query is applied on an encrypted document and corresponds to a SPARQL triple pattern. It may have an unbound subject, predicate, and/or object. The type of a query is defined using the symbols `+` and `-` which mark the bound and unbound parts, respectively. For example, a query of type `+++` requires a bound subject, a bound predicate, and a bound object. It corresponds to a SPARQL ASK query and asks whether or not the specified triple is contained in the document d. All other queries correspond to standard SPARQL queries as they can be stated in SELECT queries. For example, a query of type `-++` returns a set of subject URIs and requires a bound predicate and a bound object.

A query is created from a basic key and two query patterns. The basic key defines the type of the query and the query patterns specify the bound and unbound parts of the query. We distinguish between two different types of query patterns which are restriction patterns r and user-defined query patterns u. A restriction pattern r is defined a-priori by the data owner and narrows down the possible queries a user can perform. A user-defined query pattern u represents the query parameters specified by the user. The set of all query patterns is defined as $(\mathbb{U} \cup \{\texttt{?}\}) \times (\mathbb{U} \cup \{\texttt{?}\}) \times (\mathbb{U} \cup \mathbb{L} \cup \{\texttt{?}\})$. The variable `?` identifies unbound parts of a query pattern and corresponds to a variable in a SPARQL triple pattern like `?x`. A query pattern is defined as $(s_?, p_?, o_?)$ with $s_?$ as the

queried subject, $p_?$ the queried predicate, and $o_?$ the queried object. A query key k_q corresponds to a partially specified query which already encodes a basic key k_b and a data owner's restriction pattern r. However, a query key does not encode a user's query pattern u. Thus, a complete query is created from a query key by combining it with a user's query pattern u.

Query Functions on Encrypted Data A query function f performs a query on an encrypted document d_C and returns a result based on all matching triples t of the plaintext document d. A query function requires a query key k_q, a user's query pattern u, and the encrypted document d_C as input. Each query function supports one particular type of queries. Thus, there are eight different query functions identified as $f_{\texttt{---}}$, $f_{\texttt{+--}}$, $f_{\texttt{-+-}}$, $f_{\texttt{--+}}$, $f_{\texttt{++-}}$, $f_{\texttt{+-+}}$, $f_{\texttt{-++}}$, and $f_{\texttt{+++}}$. Again, the symbols `+` and `-` mark the bound and unbound parts of the supported queries, respectively. For example, `+` at the first position requires a subject to be specified in the query. At the second or the third position, respectively, the symbol `+` requires a predicate or an object to be specified. The result of a query function f also depends on the symbols `+` and `-`. The result can be a set of triples, a set of tuples, a set of URIs, a set of literals, or a boolean value. For example, the query function $f_{\texttt{+--}}$ returns a set of tuples (y, z) with $y \in \mathbb{U}$ and $z \in \mathbb{U} \cup \mathbb{L}$.

The bound and unbound parts of a query (s, p, o) are specified by a user's query pattern $u = (s_u, p_u, o_u)$ and the data owner's restriction pattern $r = (s_r, p_r, o_r)$, which is encoded in a query key k_q. The query patterns must comply with the type of the query function. For example, the query function $f_{\texttt{-++}}$ requires either $p_r \neq \texttt{?}$ or $p_u \neq \texttt{?}$ and either $o_r \neq \texttt{?}$ or $o_u \neq \texttt{?}$. Thus, the data owner may define a restriction pattern $r = (\texttt{?}, \texttt{rdf:type}, \texttt{?})$ which specifies `rdf:type` as the predicate of the query being performed. Then, a user can only query for instances of classes, i. e, triples with the predicate `rdf:type`. In addition, the user has to specify a query pattern $u = (\texttt{?}, \texttt{?}, o_u)$ with $o_u \neq \texttt{?}$ and cannot just leave the object position unbound (which would be possible with the query function $f_{\texttt{-+-}}$). Furthermore, a value specified in the restriction pattern r cannot be specified in the user's query pattern u as well. Thus, the data owner can restrict the possible queries a user can perform by specifying the corresponding parts in the restriction pattern r.

For example, the query function $f_{\texttt{---}}$ returns the complete plaintext document d if neither the restriction pattern $r = (s_r, p_r, o_r)$ nor the user's query pattern $u = (s_u, p_u, o_u)$ specify any particular binding, i. e., URI or literal. On the other hand, the function $f_{\texttt{+--}}$ requires either $s_r \neq \texttt{?}$ or $s_u \neq \texttt{?}$. The function searches for all triples $t = (s, p, o) \in d$ with $s = s_r$ or $s = s_u$ and returns a set of tuples (p, o). The function $f_{\texttt{++-}}$ requires both a subject and a predicate to be specified in the query. For every matching triple $t = (s, p, o) \in d$, the object o is returned. Finally, the function $f_{\texttt{+++}}$ corresponds to a SPARQL ASK query. It returns $TRUE$ iff a query (s, p, o) matches a triple in d, i. e., iff $(s, p, o) \in d$.

5 Encrypting Graph-based Documents for Querying

In order to search on encrypted graph-based documents, one first needs to prepare and encrypt the plain-text triples in an appropriate way. These tasks are part of the general process of searching on graph-based documents, which can be divided into six steps as presented Fig. 2. The first three steps correspond to the encryption phase in which a plaintext document d is encrypted. The fourth and fifth step correspond to the query preparation phase in which the data owner defines the queries to be performed on the encrypted document. In the following, the details of the first five steps are explained. The sixth step corresponds to the querying phase carried out by the user. It is presented in Section 6.

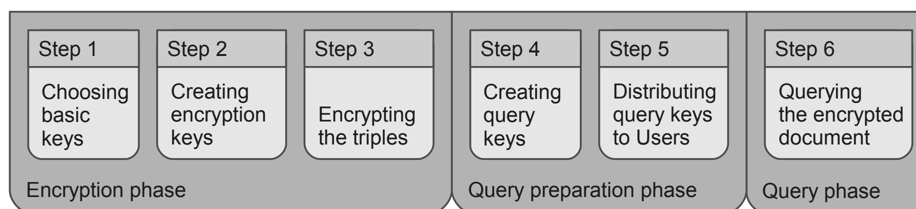


Fig. 2. Process of searching on encrypted graph-based documents.

Step 1: Choosing Basic Keys The data owner choses eight different basic keys k_b which form the foundation of creating encryption keys k_t. Each basic key k_b defines a particular query type and is thus used for one of the eight query functions f defined in Section 4. If the data owner does not define a restriction pattern r, i. e., if $r = (?, ?, ?)$, a basic key k_b is identical to a query key k_q which is provided to the user. A particular set of basic keys is only used for one plaintext document. This ensures that a user who is able to query one document will not be able to query another document using the same basic keys.

Step 2: Creating Encryption Keys The data owner creates a symmetric encryption key k_t for each plaintext triple $t \in d$ and each basic key k_b. An encryption key $k_t \in \mathbb{K}_t$ is a bit string of length l with $\mathbb{K}_t \subset \{0, 1\}^l$ being the set of all encryption keys. It encodes a basic key k_b and those parts of a triple $t = (s, p, o)$ which are required for a corresponding query. For example, the basic key k_{+--} is encoded into an encryption key k_t together with the triple's subject s. Creating an encryption key k_t requires a hash function λ and a combining function ϱ. A hash function transforms a bit string of arbitrary length to a fixed length hash value [19]. Using l as the length of the resulting bit string, the hash function λ is defined as $\lambda : \{0, 1\}^* \rightarrow \{0, 1\}^l$. The hash function is used for reducing the possibility of creating identical encryption keys for different input data. An example hash function is SHA-2 [20] with $l = 256$. The combining function ϱ combines a single bit string b and a set of n bit strings b_i with $i = 1, \ldots, n$ into a single bit string. The function is used to create the encryption key k_t based on different input data. Given a product N of two large prime numbers, the function

Table 1. Encrypting a single triple t using a basic key k_b. The symbol $||$ corresponds to the concatenation operator and ε corresponds to the empty bit string. The result is an encrypted triple $c = (c_{---}, \ldots, c_{+++})$ (see definition in Section 4).

Basic key k_b	**Creating the Encryption Keys** k_t **for a Triple** $t = (s, p, o)$	**Encrypting a Triple** $t = (s, p, o)$ **with** k_t
k_{---}	$k_t = \lambda(\varrho(k_{---}, \{\}))$	$c_{---} = \mathsf{enc}(k_t, s\|\|p\|\|o)$
k_{+--}	$k_t = \lambda(\varrho(k_{+--}, \{\alpha\|\|s\}))$	$c_{+--} = \mathsf{enc}(k_t, p\|\|o)$
...	...	...
k_{++-}	$k_t = \lambda(\varrho(k_{++-}, \{\alpha\|\|s, \beta\|\|p\}))$	$c_{++-} = \mathsf{enc}(k_t, o)$
...	...	...
k_{+++}	$k_t = \lambda(\varrho(k_{+++}, \{\alpha\|\|s, \beta\|\|p, \gamma\|\|o\}))$	$c_{+++} = \mathsf{enc}(k_t, \varepsilon)$

$\varrho : \{0,1\}^* \times 2^{\{0,1\}^*} \rightarrow \{0,1\}^*$ is defined as follows:

$$\varrho(b, \{b_1, \ldots, b_n\}) := \mathsf{bit}\left(\mathsf{int}(b)^{\prod_{i=1}^{n} \mathsf{int}(\lambda(b_i))} \bmod N\right)$$

The function int transforms a bit string into an integer value and bit transforms an integer into a bit string. In order to create an encryption key k_t with the combining function ϱ, a prefix is attached to each part of the triple t. The prefix α is used to identify the triple's subject, β identifies the triple's predicate, and γ marks the triple's object. Table 1 depicts the details of creating an encryption key k_t for different basic keys k_b.

Step 3: Encrypting the Triples The encryption key k_t is used for encrypting those parts of the triple which are not already encoded into k_t. For example, an encryption key k_t based on a basic key k_{+--} encrypts the predicate and object of a triple t. The ciphertext resulting from this operation is denoted as c_{+--}. Encrypting a triple t with an encryption key k_t is conducted using an encryption function enc. The function requires a bit string representation of the triple and the encryption key as input and returns an encrypted bit string. It is defined as $\mathsf{enc} : \mathbb{K}_t \times \{0,1\}^* \rightarrow \{0,1\}^*$. Table 1 depicts the details of encrypting a single triple t. Each triple $t \in d$ is encrypted for each of the eight basic keys k_b. This results in an encrypted triple $c = (c_{---}, c_{+--}, c_{-+-}, c_{--+}, c_{++-}, c_{+-+}, c_{-++}, c_{+++})$. The ciphertext c_{+++} is created by encrypting the empty bit string ε.

Step 4: Creating Query Keys The data owner defines the queries which can be applied on the document by creating a query key k_q for each allowed query. As defined in Section 4, a query key k_q encodes a basic key k_b and a restriction pattern $r = (s_r, p_r, o_r)$. Since a query key is used in the decryption process, its creation is similar to that of a symmetric encryption key k_t. As depicted in Table 2, the creation of a query key is also based on the combining function ϱ. The bound parts of the data owner's restriction pattern r are first associated with a prefix defining their function within a triple (see above). The prefixed strings are then combined into a query key together with the basic key by using the function ϱ. The basic key also defines the number of possible query keys to be created. For example, the basic key k_{+--} can be used for creating the query keys $k_{\mathtt{r??}}$ and $k_{\mathtt{u??}}$. The index $\mathtt{r}$ states that the corresponding part of the

Table 2. Creating a query key k_q using a basic key k_b and a data owner's restriction pattern r. Query keys are created by using a combining function ϱ.

Input		Output
Basic key k_b	**Restriction pattern r**	**Resulting query key k_q**
k_{---}	$(?,?,?)$	$k_{???} = \varrho(k_{---}, \{\})$
...	...	...
k_{++-}	$(s_r, p_r, ?), s_r, p_r \neq ?$	$k_{rr?} = \varrho(k_{++-}, \{\alpha \vert\vert s_r, \beta \vert\vert p_r\})$
k_{++-}	$(s_r, ?, ?), s_r \neq ?$	$k_{ru?} = \varrho(k_{++-}, \{\alpha \vert\vert s_r\})$
k_{++-}	$(?, p_r, ?), p_r \neq ?$	$k_{ur?} = \varrho(k_{++-}, \{\beta \vert\vert p_r\})$
k_{++-}	$(?,?,?)$	$k_{uu?} = \varrho(k_{++-}, \{\})$
...	...	...

triple is pre-defined by the data owner's restriction pattern r and `u` indicates that the user must specify the part within the query pattern u. The index `?` marks the unbound parts, which are returned as the query result.

Step 5: Distributing Query Keys to Users Finally, the data owner sends a set of query keys k_S to a user. In doing so, the data owner authorizes the user to perform the queries specified by the query keys $k_q \in k_S$. The set k_S is transmitted through a secure channel in order to ensure that only authorized users can perform the queries.

6 Performing Queries on Encrypted Documents

In the sixth step, a user queries an encrypted document d_C by decrypting each triple $c \in d_C$ with the received query keys $k_q \in k_S$. For each of the query keys, the user must define a corresponding query pattern u. As described in Section 4, the combination of a query key and the user's query pattern corresponds to a SPARQL query which is applied by a corresponding query function f. A query key encodes a data owner's restriction pattern r. The more parts a data owner specifies in the restriction pattern, the fewer choices does the user have to formulate a query. For example, the query function f_{-++} requires a predicate and an object to be either specified in r or in u. If the data owner defines $r = (?, \texttt{rdf:type}, \texttt{foaf:Person})$, a user can only search for resources of `rdf:type foaf:Person` from the FOAF [21] vocabulary. In this case, the user can only state the query $u = (?,?,?)$ adding no further constraints on the owner's restriction r. Only by this, the combination of u and r result in valid parameters for f_{-++}. However, if the data owner defines $r = (?, \texttt{rdf:type}, ?)$, the user can specify different values for the query pattern's object. For example, the user may define $u = (?, ?, \texttt{foaf:Organization})$ to search for all resources of `rdf:type foaf:Organization`. Below we first describe how a SPARQL query containing a single triple pattern is executed, before we demonstrate the application of a SPARQL query with a query group pattern.

Table 3. Creating a decryption key k'_t using a query key k_q and a user's query pattern $u = (s_u, p_u, o_u)$. The query key encodes a basic key k_b and the data owner's query restriction pattern $r = (s_r, p_r, o_r)$.

Input		Output
Query key k_q	**User query pattern** $u = (s_u, p_u, o_u)$	**User computed decryption key k'_t**
$k_{???}$	$(?, ?, ?)$	$k'_t = \lambda(\varrho(k_{???}, \{\}))$
...	...	...
$k_{\mathrm{rr?}}$	$(?, ?, ?)$	$k'_t = \lambda(\varrho(k_{\mathrm{rr?}}, \{\})$
$k_{\mathrm{ru?}}$	$(?, p_u, ?), p_u \neq ?$	$k'_t = \lambda(\varrho(k_{\mathrm{ru?}}, \{\beta \Vert p_u\}))$
$k_{\mathrm{ur?}}$	$(s_u, ?, ?), s_u \neq ?$	$k'_t = \lambda(\varrho(k_{\mathrm{ur?}}, \{\alpha \Vert s_u\}))$
$k_{\mathrm{uu?}}$	$(s_u, p_u, ?), s_u, p_u \neq ?$	$k'_t = \lambda(\varrho(k_{\mathrm{uu?}}, \{\alpha \Vert s_u, \beta \Vert p_u\}))$
...	...	...

6.1 Querying using a Single Query Pattern

A single SPARQL query is performed by applying a single query function f. This requires a query key k_q, a user's query pattern u, and an encrypted document d_C. A query function basically tries to decrypt each encrypted triple $c \in d_C$. This process can be further subdivided into three sub-steps. In the first sub-step, the decryption key is created which is used in the second sub-step to decrypt the encrypted triples. If the decryption is successful, the corresponding plaintext triple $t \in d$ will be used for creating the query result in the third sub-step.

Step 6a: Computing Decryption Keys A decryption key k'_t is created similar to the encryption key k_t. It encodes a query key k_q and the user's query pattern $u = (s_u, p_u, o_u)$. The number of possible combinations of different query keys k_q and query patterns u depend on the query function f. For example, a query function $f_{\text{++-}}$ requires that the subject and object of the query is either defined by the user's query pattern u or encoded as a query restriction in the query key. Table 3 depicts the creation of a decryption key k'_t based on the query key k_q and the user's query pattern u.

Step 6b: Decrypting the Triples The query function tries to decrypt each encrypted triple $c \in d_C$ using a decryption key $k_{t'} \in \mathbb{K}_t$ and a decryption function dec. The function is similar to the encryption function enc. It requires a decryption key k'_t and a bit string as input and returns a decrypted bit string as output. The decryption function is defined as $\mathsf{dec} : \mathbb{K}_t \times \{0,1\}^* \to \{0,1\}^*$. If the decryption key k'_t was created correctly, it is $k_t = k'_t$. In this case, the decryption process is inverse to the encryption operation, i. e., $b = \mathsf{dec}(k'_t, \mathsf{enc}(k_t, b))$ with b being a bit string representation of the plaintext triples t.

Step 6c: Creating the Query Result The created query result corresponds to the output of a query function f. Creating this output is based on the decryption function dec. If the decryption process of an encrypted triple c is successful, the corresponding plaintext triple t fulfills the specified query. If the triple does not fulfill the query, it is $k_t \neq k'_t$ and $b \neq \mathsf{dec}(k'_t, \mathsf{enc}(k_t, b))$. In this case, the triple t is not part of the query result.

6.2 Applying SPARQL Queries with Multiple Query Patterns

A complex SPARQL query with n triple patterns is conducted by performing a separate query for each of the n patterns and combining their results. The query corresponds to a set k_S of n query keys k_i with $i = 1 \dots n$. First, all queries of type +++ are applied. Such queries correspond to SPARQL ASK queries. If at least one ASK query is not satisfied, the whole query cannot be satisfied. In this case, the querying can already be stopped. After applying an ASK query, the corresponding query key is removed from k_S. Thus, all query keys k_i remaining after the first step contain at least one unbound variable. Second, the remaining queries are applied as well. The result of each query contains all possible bindings of the query's variables. It can be interpreted as a table with its columns being the variables of the query and its rows being the different solutions [22]. The result tables of all queries are incrementally combined using the join operator $\bowtie$ [22]. Finally, the complete query result is returned.

7 Implementation and Evaluation

We have implemented our approach by using established cryptographic algorithms. For the encryption function enc, we use the Advanced Encryption Standard (AES) [23] with a key length of 256 bits. For the hash function λ, we use the Secure Hash Algorithm 2 (SHA-2) [20] with an output length of 256 bits. As feature by design, the length of the hash value and the key length of the encryption function are the same. Thus, the computed hash value can directly be used as encryption key. The combining function ϱ is based on the RSA algorithm. In our implementation, we use a value of N with 2048 bits [24].

As first evaluation of our approach, we have used the Berlin SPARQL benchmark (BSBM) [25] which provides tools for generating synthetic graph data and SPARQL queries that simulate a user searching the data. We have used the tools to create three different RDF graphs for our experiment with $5 \cdot 10^4$, 10^5, and $2 \cdot 10^5$ triples, respectively. Encrypting a document with $5 \cdot 10^4$ triples takes about 56 minutes. Increasing the size of the plaintext document by a certain factor also increases the time required for its encryption by the same factor. The runtime of each query depends on the number of triples in the document and on the number of triple patterns in the query. For example, evaluating a query with n triple patterns for $m \cdot 10^5$ triples takes about $n \cdot m \cdot 950$ ms. This constant factor is due to the design of our query algorithm as described in Section 6.

8 Conclusion and Future Work

We have presented an approach for searching on encrypted graph data. Queries are authorized by the owner of the encrypted data who can further restrict the allowed queries. A query is applied by decrypting the encrypted triples with a query key received from the data owner. Our approach supports a subset of the SPARQL query language including queries of type SELECT and ASK.

References

1. Knechtel, M., Stuckenschmidt, H.: Query-based access control for ontologies. In: Web Reasoning and Rule Systems, Springer (2010) 73–87
2. Bellare, M., Boldyreva, A., O'Neill, A.: Deterministic and efficiently searchable encryption. In: CRYPTO. Springer (2007) 535–552
3. Scholl, M., Stine, K., Hash, J., Bowen, P., Johnson, A., Smith, C.D., Steinberg, D.: An introductory resource guide for implementing the HIPAA security rule (2008)
4. Cao, X., Klusch, M.: Dynamic semantic data replication for k-random search in peer-to-peer networks. In: NCA 2012, IEEE (2012) 20–27
5. Boneh, D., Sahai, A., Waters, B.: Functional encryption: A new vision for public-key cryptography. CACM **55**(11) (2012) 56–64
6. Abdalla, M., Bellare, M., Catalano, D., Kiltz, E., Kohno, T., Lange, T., Malone-Lee, J., Neven, G., Paillier, P., Shi, H.: Searchable encryption revisited. In: CRYPTO. (2005) 205–222
7. Ding, L., Finin, T.: Characterizing the semantic web on the web. In: ISWC, Springer (2006) 242–257
8. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C (2008) `http://www.w3.org/TR/rdf-sparql-query/`.
9. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: S&P 2000, IEEE (2000) 44–55
10. Goh, E.J.: Secure indexes. IACR Cryptology ePrint Archive **2003** (2003) 216
11. Sahai, A., Waters, B.: Fuzzy identity-based encryption. In: EUROCRYPT 2005, Springer (2005) 457–473
12. O'Neill, A.: Definitional issues in functional encryption. IACR Cryptology ePrint Archive **2010** (2010) 556
13. Boneh, D., Sahai, A., Waters, B.: Functional encryption: Definitions and challenges. Theory of Cryptography **6597** (2011) 253–273
14. Poh, G.S., Mohamad, M.S., Z'aba, M.R.: Structured encryption for conceptual graphs. In: IWSEC. (2012) 105–122
15. Kamara, S., Wei, L.: Garbled circuits via structured encryption. In: WAHC. (2013)
16. Popa, R.A., Redfield, C.M.S., Zeldovich, N., Balakrishnan, H.: Cryptdb: protecting confidentiality with encrypted query processing. In: SOSP. (2011) 85–100
17. Wang, W.H., Lakshmanan, L.V.S.: Efficient secure query evaluation over encrypted XML databases. In: VLDB. (2006) 127–138
18. Brinkman, R., Feng, L., Doumen, J., Hartel, P.H., Jonker, W.: Efficient tree search in encrypted data. Information Systems Security **13**(3) (2004) 14–21
19. Rogaway, P., Shrimpton, T.: Cryptographic hash-function basics. In: FSE. Volume 3017., Springer (2004) 371–388
20. NIST: Secure hash standard. FIPS PUB 180-4 (03 2012) `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`.
21. Brickley, D., Miller, L.: FOAF vocabulary specification (2010) `http://xmlns.com/foaf/spec/`.
22. Cyganiak, R.: A relational algebra for SPARQL. Technical report, HP Labs (2005)
23. NIST: Advanced encryption standard. FIPS PUB 197 (2001) `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.
24. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. CACM **21**(2) (1978) 120–126
25. Bizer, C., Schultz, A.: The berlin SPARQL benchmark. IJSWIS **5**(2) (2009) 1–24