Self-Enforcing Access Control for Encrypted RDF *

Javier D. Fernández[1,2], Sabrina Kirrane[1], Axel Polleres[1,2], and Simon Steyskal[1,3]

[1] Vienna University of Economics and Business, Vienna, Austria
[firstname.lastname]@wu.ac.at
[2] Complexity Science Hub Vienna, Vienna, Austria
[3] Siemens AG Österreich, Vienna, Austria

Abstract. The amount of raw data exchanged via web protocols is steadily increasing. Although the Linked Data infrastructure could potentially be used to selectively share RDF data with different individuals or organisations, the primary focus remains on the unrestricted sharing of public data. In order to extend the Linked Data paradigm to cater for closed data, there is a need to augment the existing infrastructure with robust security mechanisms. At the most basic level both access control and encryption mechanisms are required. In this paper, we propose a flexible and dynamic mechanism for securely storing and efficiently querying RDF datasets. By employing an encryption strategy based on Functional Encryption (FE) in which controlled data access does not require a trusted mediator, but is instead enforced by the cryptographic approach itself, we allow for fine-grained access control over encrypted RDF data while at the same time reducing the administrative overhead associated with access control management.

1 Introduction

The Linked Data infrastructure could potentially be used not only to distributedly share public data, but also to selectively share data, perhaps of a sensitive nature (e.g., personal data, health data, financial data, etc.), with specific individuals or organisations (i.e., closed data). In order to realise this vision, we must first extend the existing Linked Data infrastructure with suitable security mechanisms. More specifically, encryption is needed to protect data in case the server is compromised, while access control is needed to ensure that only authorised individuals can access specific data. Apart from the need to protect data, robustness in terms of usability, performance, and scalability is a major consideration.

However, current encryption techniques for RDF are still very limited, especially with respect to the flexible maintenance and querying of encrypted data in light of user access control policies. Initial partial encryption techniques [16, 17] focus on catering for both plain and encrypted data in the same representation

* Supported by the Austrian Science Fund (FWF): M1720-G11, the Austrian Research Promotion Agency (FFG) under grant 845638, and European Union's Horizon 2020 research and innovation programme under grant 731601.

and how to incorporate the metadata necessary for decryption. More recently, [21] proposed the generation of multiple ciphertexts per triple (i.e. each triple is encrypted multiple times depending on whether or not access to the subject, predicate and/or object is restricted) and the distribution of several keys to users. Although finer-grained access control is supported, the maintenance of multiple ciphertexts (i.e. encrypted triples) and keys presents scalability challenges. Additionally, such an approach, or likewise term-based encryption of RDF graphs, means that the *structure* of parts of the graph that should not be accessible could potentially be recovered, thus posing a security risk (cf. for instance [32]).

Beyond RDF, novel cryptography mechanisms have been developed that enable the flexible specification and enforcement of access policies over encrypted data. Predicate-based Encryption (PBE) [22] – which we refer to as Functional Encryption (FE) in order to avoid confusion with *RDF predicates* – enables searching over encrypted data, mainly for keywords or the conjunction of keyword queries, while alleviating the re-encryption burden associated with adding additional data.

Herein, we extend recent findings on FE to RDF, and demonstrate how FE can be used for fine-grained access control based on triples patterns over encrypted RDF datasets. Summarising our contributions, we: (i) adapt functional encryption to RDF such that it is possible to enforce access control over encrypted RDF data in a self-enforcing manner; (ii) demonstrate how encryption keys based on triple patterns can be used to specify flexible access control for Linked Data sources; and (iii) propose and evaluate indexing strategies that enhance query performance and scalability. Experiments show reasonable loading and query performance overheads with respect to traditional, non-encrypted data retrieval. The remainder of the paper is structured as follows: We discuss related work and potential alternatives to our proposal in *Section* 2. The details of our specific approach and optimisations are presented in *Section* 3 and *Section* 4 respectively, and evaluated in *Section* 5. Finally, we conclude and outline directions for future work in *Section* 6.

2 Related Work

When it comes to access control for RDF, broadly speaking researchers have focused on representing existing access control models and standards using semantic technology; proposing new access control models suitable for open, heterogeneous and distributed environments; and devising languages and frameworks that can be used to facilitate access control policy specification and maintenance. Kirrane et al. [23] provide a comprehensive survey of existing access control proposals for RDF. Unlike access control, encryption techniques for RDF has received very little attention to date. Giereth [17] demonstrate how public-key encryption techniques can be used to partially encryption RDF data represented using XML. While, Giereth [17] and Gerbracht [16] propose strategies for combining partially encrypted RDF data with the metadata that is necessary for decryption. Kasten et al. [21] propose a framework that can be used to query encrypted data. In order to support SPARQL queries based on triple patterns each triple is encrypted eight times according to the eight different binding pos-

sibilities. Limitations of the approach include the blowup associated with maintaining eight ciphers per triple and the fact that the structure of the graph is still accessible.

Searchable Symmetric Encryption (SSE) [9] has been extensively applied in database-as-a-service and cloud environments. SSE techniques focus on the encryption of outsourced data such that an external user can encrypt their query and subsequently evaluate it against the encrypted data. More specifically, SSE extracts the key features of a query (the data structures that allow for its resolution) and encrypts them such that it can be efficiently evaluated on the encrypted data. Extensive work has been done in basic SSE, which caters for a single keyword [6]. Recent improvements have been proposed to handle conjunctive search over multiple keywords [4], and to optimise the resolution to cater for large scale data in the presence of updates [5, 20, 30]. However, all of these works focus on keyword-based retrieval, whereas structured querying (such as SPARQL) over encrypted RDF datasets would require (at least) an unrestricted set of triple query patterns. In contrast, Fully Homomorphic Encryption (FHE) [15] allows any general circuit/computation over encrypted data, however it is prohibitively slow for most operations [7, 28]. Thus, practical, encryption databases such as CryptDB [28] make use of lighter forms of encryption that still cater for computations (such as sums) over the encrypted data [27], at the cost of different vulnerability/feasibility trade-offs. Recently, predicate encryption [22], whereby predicates correspond to the evaluation of disjunctions, polynomial equations and inner products, enables security in light of unrestricted queries. Predicate encryption has has a proven track record of efficiency in terms of conjunctive equality, range and subset queries.

The solution we propose builds on an existing work that defines access control policies based on RDF patterns that are in turn enforced over RDF datasets [23]. While existing proposals enforce access control over plain RDF data via data filtering (i.e., a query is executed against a dataset which is generated by removing the unauthorised data) or query rewriting (i.e., a query is updated so that unauthorised data will not be returned and subsequently executed over the unmodified dataset), we demonstrate how functional encryption can be used to enforce access control over encrypted RDF data in a self-enforcing manner (i.e., without the need for either data filtering or query rewriting nor a trusted mediator). Unlike previous approaches we store one cipher per triple and employ indexing strategies based on secure hashes (cf. PBKDF2 [19]) that can be used for efficient querying of encrypted RDF. In addition, we propose a mechanism to obfuscate the graph structure with real indexes and dummy ciphers that cannot be decrypted, making the dummy hashes and ciphers indistinguishable from real hashes and ciphers.

3 Secure and Fine-grained Encryption of RDF

Common public-key encryption schemes usually follow an all-or-nothing approach (i.e., given a particular decryption key, a ciphertext can either be decrypted or not) which in turn requires users to manage a large amount of keys,

especially if there is a need for more granular data encryption [2]. Recent advances in public-key cryptography, however, have led to a new family of encryption schemes called *Functional Encryption (FE)* which addresses aforementioned issue by making encrypted data self-enforce its access restrictions, hence, allowing for fine-grained access over encrypted information. In a functional encryption scheme, each decryption key is associated with a boolean function and each ciphertext is associated with an element of some attribute space Σ; a decryption key corresponding to a boolean function f is able to decrypt a particular ciphertext associated with $I \in \Sigma$ iff $f(I) = 1$. A functional encryption scheme is defined as a tuple of four distinct algorithms (**Setup**, **Enc**, **KeyGen**, **Dec**) such that:

Setup is used for generating a master public and master secret key pair.
Enc encrypts a plaintext message m given the master public key and an element $I \in \Sigma$. It returns a ciphertext c.
KeyGen takes as input the master secret key and generates a decryption key (i.e., secret key) SK_f for a given boolean function f.
Dec takes as input a secret key SK_f and a ciphertext c. It extracts I from c and computes $f(I)$.

3.1 A Functional Encryption Scheme for RDF

While there exist various different approaches for realising functional encryption schemes, we build upon the work of Katz et al. [22] in which functions correspond to the computation of inner-products over $\mathbb{Z}_N$ (for some large integer N). In their construction, they use $\Sigma = \mathbb{Z}_N^n$ as set of possible ciphertext attributes of length n and $\mathcal{F} = \{f_{\vec{x}} | \vec{x} \in \mathbb{Z}_N^n\}$ as the class of decryption key functions. Each ciphertext is associated with a (secret) attribute vector $\vec{y} \in \Sigma$ and each decryption key corresponds to a vector $\vec{x}$ that is incorporated into its respective boolean function $f_{\vec{x}} \in \mathcal{F}$ where $f_{\vec{x}}(\vec{y}) = 1$ iff $\sum_{i=1}^{n} y_i x_i = 0$.

In the following, we discuss how this encryption scheme can be utilised (i.e., its algorithms adopted[4]) to provide fine-grained access over encrypted RDF triples. Thus, allow for querying encrypted RDF using triple patterns such that a particular decryption key can decrypt all triples that satisfy a particular triple pattern (i.e., one key can open multiple locks). For example, a decryption key generated from a triple pattern `(?,p,?)` should be able to decrypt all triples with `p` in the predicate position.

Encrypting RDF Triples (Enc) To be able to efficiently encrypt large RDF datasets, we adopt a strategy commonly used in public-key infrastructures for securely and efficiently encrypting large amounts of data called *Key Encapsulation* [24]. Key encapsulation allows for secure but slow asymmetric encryption to be combined with simple but fast symmetric encryption by using asymmetric encryption algorithms for deriving a symmetric encryption key (usually in terms of a seed) which is subsequently used by encryption algorithms such as AES [11] for the actual encryption of the data. We illustrate this process in Figure 1.

[4] The **Setup** algorithm remains unchanged.

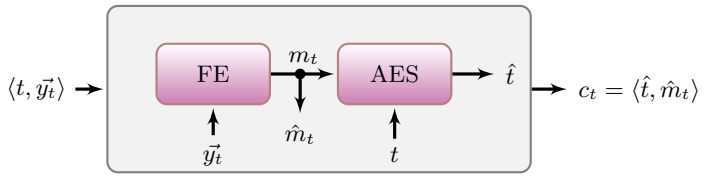

Fig. 1: Process of encrypting an RDF triple t.

Thus, to encrypt an RDF triple $t = (s, p, o)$, we first compute its respective triple vector (i.e., attribute vector) $\vec{y_t}$ and functionally encrypt (i.e., compute **Enc** as defined in [22]) a randomly generated seed m_t using $\vec{y_t}$ as the associated attribute vector. Triple vector $\vec{y_t}$ where $\vec{y_t} = (y_s, y'_s, y_p, y'_p, y_o, y'_o)$ for triple t is constructed as follows, where σ denotes a mapping function that maps a triple's subject, predicate, and object value to elements in $\mathbb{Z}_N$:

$$y_l := -r \cdot \sigma(l),\ y'_l := r, \text{ with } l \in \{s, p, o\} \text{ and random } r \in \mathbb{Z}_N$$

Table 1 illustrates the construction of a triple vector $\vec{y_t}$ based on RDF triple t.

Triple t	Triple Vector $\vec{y_t}$
t_1 = (s$_1$,p$_1$,o$_1$)	$\vec{y}_{t_1} = (-r_1 \cdot \sigma(s_1), r_1, -r_2 \cdot \sigma(p_1), r_2, -r_3 \cdot \sigma(o_1), r_3)$
t_2 = (s$_2$,p$_2$,o$_2$)	$\vec{y}_{t_2} = (-r_4 \cdot \sigma(s_2), r_4, -r_5 \cdot \sigma(p_2), r_5, -r_6 \cdot \sigma(o_2), r_6)$
...	...
t_n = (s$_n$,p$_n$,o$_n$)	$\vec{y}_{t_n} = (-r_{3n-2} \cdot \sigma(s_n), r_{3n-2}, -r_{3n-1} \cdot \sigma(p_n), r_{3n-1}, -r_{3n} \cdot \sigma(o_n), r_{3n})$

Table 1: Computing the triple vector $\vec{y_t}$ of an RDF triple t.

We use AES to encrypt the actual plaintext triple t with an encryption key derivable from our previously generated seed m_t and return both, the resulting AES ciphertext of t denoted by $\hat{t}$ and the ciphertext of the seed denoted by $\hat{m}_t$ as final ciphertext triple $c_t = \langle \hat{t}, \hat{m}_t \rangle$.

Generating Decryption Keys (KeyGen) As outlined above, decryption keys must be able to decrypt all triples that satisfy their inherent triple pattern (i.e., one query key can open multiple locks). In order to compute a decryption key based on a triple pattern $tp = (s, p, o)$ with s, p, and o either bound or unbound, we define its corresponding vector $\vec{x}$ as $\vec{x}_{tp} = (x_s, x'_s, x_p, x'_p, x_o, x'_o)$ with:

$$\text{if } l \text{ is bound: } x_l := 1, x'_l := \sigma(l), \text{ with } l \in \{s, p, o\}$$
$$\text{if } l \text{ is not bound: } x_l := 0, x'_l := 0, \text{ with } l \in \{s, p, o\}$$

Again, σ denotes a mapping function that maps a triple pattern's subject, predicate, and object value to elements in $\mathbb{Z}_N$. Table 2 illustrates the construction of a query vector $\vec{x}_{tp}$ that corresponds to a triple pattern tp.

Decryption of RDF Triples (Dec) To verify whether an encrypted triple can be decrypted with a given decryption key, we compute the inner-product of their corresponding triple vector $\vec{y_t}$ and query vector $\vec{x}_{tp}$, with $t = (s_t, p_t, o_t)$ and $tp = (s_{tp}, p_{tp}, o_{tp})$:

Triple Pattern tp	Query Vector $\vec{x}_{tp}$
tp_1 = (?,?,?)	$\vec{x}_{tp_1} = (0, 0, 0, 0, 0, 0)$
tp_2 = (s_2,?,?)	$\vec{x}_{tp_2} = (1, \sigma(s_2), 0, 0, 0, 0)$
tp_3 = (s_3,p_3,?)	$\vec{x}_{tp_3} = (1, \sigma(s_3), 1, \sigma(p_3), 0, 0)$
...	...
tp_n = (s_n,p_n,o_n)	$\vec{x}_{tp_n} = (1, \sigma(s_n), 1, \sigma(p_n), 1, \sigma(o_n))$

Table 2: Computing the query vector $\vec{x}_{tp}$ that corresponds to a triple pattern tp

$$\vec{y}_t \cdot \vec{x}_{tp} = y_{s_t} x_{s_{tp}} + y'_{s_t} x'_{s_{tp}} + y_{p_t} x_{p_{tp}} + y'_{p_t} x'_{p_{tp}} + y_{o_t} x_{o_{tp}} + y'_{o_t} x'_{o_{tp}}$$

Only when $\vec{y}_t \cdot \vec{x}_{tp} = 0$ is it possible to decrypt the encrypted seed $\hat{m}_t$, hence the corresponding symmetric AES key can be correctly derived and the plaintext triple t be returned. Otherwise (i.e., $\vec{y}_t \cdot \vec{x}_{tp} \neq 0$), an arbitrary seed $m' \neq m_t$ is generated hence encrypted triple c_t cannot be decrypted [26].

4 Optimising Query Execution over Encrypted RDF

The *secure data store* holds all the encrypted triples, i.e. $\{c_{t_1}, c_{t_2}, \cdots, c_{t_n}\}$, being n the total number of triples in the dataset. Besides assuring the confidentiality of the data, the data store is responsible for enabling the querying of encrypted data.

In the most basic scenario, since triples are stored in their encrypted form, a user's query would be resolved by iterating over all triples in the dataset, checking whether any of them can be decrypted with a given decryption key. Obviously, this results in an inefficient process at large scale. As a first improvement one can distribute the set of encrypted triples among different peers such that decryption could run in parallel. In spite of inherent performance improvements, such a solution is still dominated by the available number of peers and the – potentially large – number of encrypted triples each peer would have to process. Current efficient solutions for querying encrypted data are based on (a) using indexes to speed up the decryption process by reducing the set of potential solutions; or (b) making use of specific encryption schemes that support the execution of operations directly over encrypted data [13]. Our solution herein follows the first approach, whereas the use of alternative and directly encryption mechanisms (such as homomorphic encryption [28]) is complementary and left to future work.

In our implementation of such a secure data store, we first encrypt all triples and store them in a key-value structure, referred to as an `EncTriples Index`, where the keys are unique integer IDs and the values hold the encrypted triples (see Figure 2 and Figure 3 (right)). Note that this structure can be implemented with any traditional *Map* structure, as it only requires fast access to the encrypted value associated with a given ID. In the following, we describe two alternative approaches, i.e., one using *three individual indexes* and one based on *Vertical Partitioning (VP)* for finding the range of IDs in the `EncTriples Index` which can satisfy a triple pattern query. In order to maintain simplicity and general applicability of the proposed store, both alternatives consider key-value backends, which are increasingly used to manage RDF data [8], especially in distributed scenarios. It is also worth mentioning that we focus on basic triple

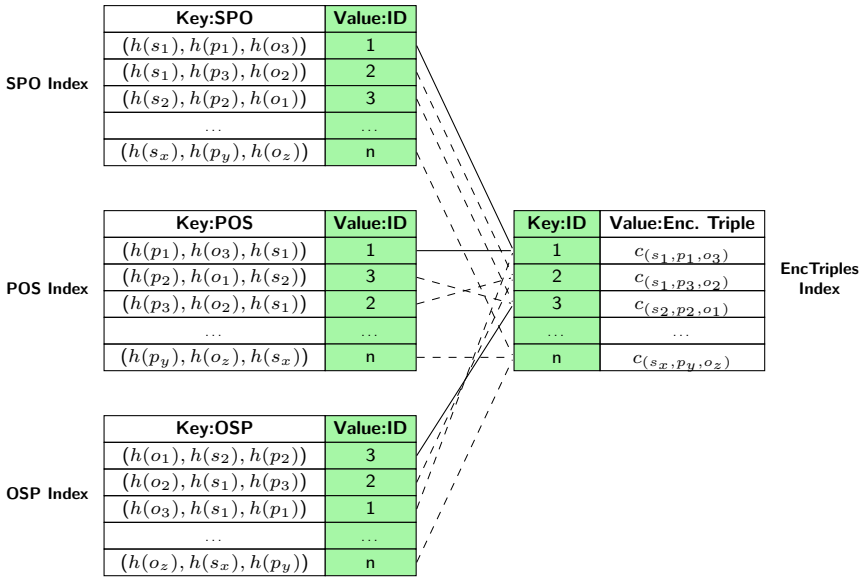

Fig. 2: `3-Index` approach for indexing and retrieval of encrypted triples.

pattern queries as (i) they are the cornerstone that can be used to build more complex SPARQL queries, and (ii) they constitute all the functionality to support the Triple Pattern Fragments [31] interface.

3-Index Approach. Following well-known indexing strategies, such as from CumulusRDF [25], we use three key-value B-Trees in order to cover all triple pattern combinations: `SPO`, `POS` and `OSP Indexes`. Figure 2 illustrates this organisation. As can be seen, each index consists of a *Map* whose keys are the securely hashed (cf. PBKDF2 [19]) subject, predicate, and object of each triple, and values point to IDs storing the respective ciphertext triples in the `EncTriples Index`.

Algorithm 1 shows the resolution of a `(s,p,o)` triple pattern query using the `3-Index` approach. First, we compute the secure hashes `h(s)`, `h(p)` and `h(o)` from the corresponding `s`, `p` and `o` provided by the user (Line 1). Our $hash(s, p, o)$ function does not hash unbounded terms in the triple pattern but treats them as a wildcard '`?`' term (hence all terms will be retrieved in the subsequent range queries). Then, we select the best index to evaluate the query (Line 2). In our case, the `SPO Index` serves `(s,?,?)` and `(s,p,?)` triple patterns, the `POS Index` satisfies `(?,p,?)` and `(?,p,o)`, and the `OSP Index` index serves `(s,?,o)` and `(?,?,o)`. Both `(s,p,o)` and `(?,?,?)` can be solved by any of them. Then, we make use of the selected index to get the range of values where the given `h(s), h(p), h(o)` (or 'anything' if the wildcard '`?`' is present in a term) is stored (Line 3). Note that this search can be implemented by utilising B-Trees [10, 29] for indexing the keys. For each of the candidate ID values in the range (Line 4), we retrieve the encrypted triple for such ID by searching for this ID in the `EncTriples Index` (Line 5). Finally, we proceed with the decryption of

Algorithm 1 `3-Index_Search(s,p,o,key)`

```
1: (h(s), h(p), h(o)) ← hash(s, p, o);
2: index ← selectBestIndex(s, p, o);                    ▷ index = {SPO|POS|OSP}
3: IDs[ ] ← index.getRangeValues(h(s), h(p), h(o));
4: for each (id ∈ IDs) do
5:     encryptedTriple ← EncTriples.get(id);
6:     < decryptedTriple, status >← Decrypt(encryptedTriple, key);
7:     if (status = valid) then
8:         output(decryptedTriple);
9:     end if
10: end for
```

the encrypted triple using the `key` provided by the user (Line 6). If the status of such decryption is *valid* (Line 7) then the decryption was successful and we output the decrypted triples (Line 8) that satisfy the query.

Thus, the combination of the three `SPO`, `POS` and `OSP Indexes` reduces the search space of the query requests by applying simple range scans over hashed triples. This efficient retrieval has been traditionally served through tree-based map structures guaranteeing $log(n)$ costs for searches and updates on the data, hence we rely on B-Tree stores for our practical materialisation of the indexes. In contrast, supporting all triple pattern combinations in `3-Index` comes at the expense of additional space overheads, given that each `(h(s),h(p),h(o))` of a triple is stored three times (in each `SPO`, `POS` and `OSP Indexes`). Note, however, that this is a typical scenario for RDF stores and in our case the triples are encrypted and stored just once (in `EncTriples Index`).

Vertical Partitioning Approach. Vertical partitioning [1] is a well-known RDF indexing technique motivated by the fact that usually only a few predicates are used to describe a dataset [14]. Thus, this technique stores one "table" per predicate, indexing `(S,O)` pairs that are related via the predicate. In our case, we propose to use one key-value B-Tree for each `h(p)`, storing `(h(s),h(o))` pairs as keys, and the corresponding ID as the value. Similar to the previous case, the only requirement is to allow for fast range queries on their map index keys. However, in the case of an `SO` index, traditional key-value schemes are not efficient for queries where the first component (the subject) is unbound. Thus, to improve efficiency for triple patterns with unbounded subject (i.e. `(?,`p_y`,`o_z`)` and `(?,?,`o_z`)`), while remaining in a general key-value scheme, we duplicate the pairs and introduce the inverse `(h(o),h(s))` pairs. The final organisation is shown in Figure 3 (left), where the predicate maps are referred to as `Pred_h(`p_1`)`, `Pred_h(`p_2`)`,..., `Pred_h(`p_n`) Indexes`. As depicted, we add `"so"` and `"os"` keywords to the stored composite keys in order to distinguish the order of the key.

Algorithm 2 shows the resolution of a `(s,p,o)` triple pattern query with the `VP` organisation. In this case, after performing the variable initialisation (Line 1) and the aforementioned secure hash of the terms (Line 2), we inspect the predicate term `h(p)` and select the corresponding predicate index (Line 3), i.e., `Pred_h(p)`. Nonetheless, if the predicate is unbounded, all predicate indexes are selected as we have to iterate through all tables, which penalises the performance

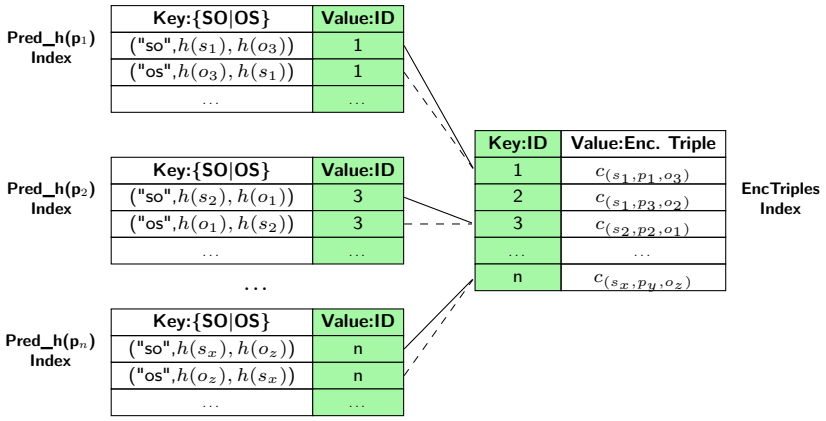

Fig. 3: `Vertical Partitioning (VP)` approach for indexing and retrieval of encrypted triples.

Algorithm 2 `VerticalPartitioning_Search(s,p,o,key)`

```
1: IDs[ ] ← ();
2: (h(s), h(p), h(o)) ← hash(s, p, o);
3: Indexes[] ← selectPredIndex(h(p)); ▷ Indexes ⊆ {Pred_h(p1), ··· , Pred_h(pn)Index}
4: for each (index ∈ Indexes) do
5:     if (s =?) then
6:         IDs[ ] ← index.getRangeValues("os", h(o), ?);
7:     else
8:         IDs[ ] ← index.getRangeValues("so", h(s), h(o));
9:     end if
10:    for each (id ∈ IDs) do
11:        encryptedTriple ← EncTriples.get(id);
12:        < decryptedTriple, status >← Decrypt(encryptedTriple, key);
13:        if (status = valid) then
14:            output(decryptedTriple);
15:        end if
16:    end for
17: end for
```

of such queries. For each predicate index, we then inspect the subject term (Lines 5-9). If the subject is unbounded (Line 5), we will perform a `("os",h(o),?)` range query over the corresponding predicate index (Line 6), otherwise we execute a `("so",h(s),h(o))` range query. Note that in both cases the object could also be unbounded. The algorithm iterates over the candidates IDs (Lines 10-end) in a similar way to the previous cases, i.e., retrieving the encrypted triple from `EncTriples Index` (Line 11) and performing the decryption (Lines 12-14).

Overall, `VP` needs less space than the previous `3-Index` approach, since the predicates are represented implicitly and the subjects and objects are represented only twice. In contrast, it penalises the queries with unbound predicate as it has to iterate through all tables. Nevertheless, studies on SPARQL query logs show that these queries are infrequent in real applications [3].

Protecting the Structure of Encrypted Data. The proposed hash-based indexes are a cornerstone for boosting query resolution performance by reducing

Dataset	Triples	|S|	|P|	|O|	Size (MB)
Census	361,842	51,768	26	6,901	52
Jamendo	1,049,637	335,925	26	440,602	144
AEMET	3,547,154	394,289	23	793,664	726
LUBM	100,000	22,932	18	11,588	15
	200,000	39,244	18	23,749	29
	500,000	87,984	18	60,028	71
	1,000,000	169,783	18	120,464	139
	2,000,000	333,105	18	241,342	277
	5,000,000	820,185	18	604,308	694

Table 3: Statistical dataset description.

the encrypted candidate triples that may satisfy the user queries. The use of secure hashes [19] assures that the terms cannot be revealed but, in contrast, the indexes themselves reproduce the structure of the underlying graph (i.e., the in/out degree of nodes). However, the structure should also be protected as hash-based indexes can represent a security risk if the data server is compromised. State-of-the-art solutions (cf., [13]) propose the inclusion of spurious information, that the query processor must filter out in order to obtain the final query result.

In our particular case, this technique can be adopted by adding dummy triple hashes into the indexes with a corresponding ciphertext (in `EncTriples Index`) that cannot be decrypted by any key, hence will not influence the query results. Such an approach ensures that both the triple hashes and their corresponding ciphertexts are not distinguishable from real data.

5 Evaluation

We develop a prototypical implementation[5] of the proposed encryption and indexing strategies. Our tool is written in Java and it relies on the Java Pairing-Based Cryptography Library (JPBC [12]) to perform all the encryption/decryption operations. While, we use MapDB[6] as the supporting framework for the indexes. We provide an interface that takes as input a triple pattern query and a query key, and outputs the results of the query.

We evaluate our proposal in two related tasks: (i) performance of the data loading (encryption and indexing) and (ii) performance of different user queries (query execution on encrypted data). In both cases, we compare our proposed `3-Index` strategy w.r.t the vertical partitioning (`VP`) approach. Finally, we measure the performance overhead associated with query resolution, introduced by the secure infrastructure, by comparing its results with a counterpart non-secure triplestore. For a fair comparison, we implement the non-secure triplestore with similar `3-Index` and `VP` indexing strategies, storing the RDF data in plain. The approaches are referred to as `3-Index-plain` and `VP-plain` respectively.

Table 3 describes our experimental datasets, reporting the number of triples, different subjects (|S|), predicates (|P|) and objects (|O|), as well as the file size (in NT format). Note that there is no standard RDF corpus that can be used to

[5] Source code and experimental datasets are available at: https://aic.ai.wu.ac.at/comcrypt/sld/.

[6] http://www.mapdb.org/

evaluate RDF encryption approaches, hence we choose a diverse set of datasets that have been previously used to benchmark traditional RDF stores or there is a use case that indicates they could potentially benefit from a secure data store. On the one hand, we use the well-known Lehigh University Benchmark (LUBM [18]) data generator to obtain synthetic datasets of incremental sizes from 100K triples to 5M triples. On the other hand, we choose real-world datasets from different domains: `Census` represents the 2010 Australian census, where sensitive data must be preserved and users could have different partial views on the dataset; `Jamendo` lists music records and artists, where some data can be restricted to certain subscribers; and `AEMET` includes sensor data from weather stations in Spain, which is a real use case where the old data is public but the most recent data is restricted to particular users. Tests were performed on a computer with 2 x Intel Xeon E5-2650v2 @ 2.6 GHz (16 cores), RAM 171 GB, 4 HDDs in RAID 5 config. (2.7 TB netto storage), Ubuntu 14.04.5 LTS running on a VM with QEMU/KVM hypervisor. All of the reported (elapsed) times are the average of three independent executions.

Data loading. Figure 4 shows the dataset load times[7] for the `3-Index` and `VP` strategies. The reported time consists of the time to encrypt the triples using the aforementioned FE scheme, and the time to securely hash the terms and create the different indexes. In contrast, the non-secure triplestores, i.e. the `3-Index-plain` and `VP-plain` counterparts, only require the dataset to be indexed (we also make use of the hash of the terms in order to compare the encryption overhead).

The results show that the time of both the `3-Index` and the `VP` strategy scales linearly with the number of triples, which indicates that the representation can scale in the envisioned Linked Data scenario. It is worth noting that both strategies report similar performance results, where `VP` is slightly faster for loading given that only the subject and object is used to index each triple (the predicate is implicitly given by vertical partitioning). Finally, note that the comparison w.r.t the plain counterparts shows that the encryption overhead can be of one order of magnitude for the smaller datasets. In contrast, the encryption overhead is greatly reduced for larger datasets which is primarily due to the fact that the loading time for large datasets is the predominant factor, as the B-Tree indexes become slower the more triples are added (due to rebalancing).

Query resolution. Figure 5 shows the query resolution time for two selected datasets[8], LUBM with 5M triples and Jamendo, considering all types of triple patterns. To do so, we sample 1,000 queries of each type and report the average resolution time. As expected, the `3-Index` reports a noticeable better performance than `VP` for queries with unbound predicates given that `VP` has to iterate though all predicate tables in this case. In turn, the `3-Index` and the `VP` approaches remain competitive with respect to their non-secure counterparts, if a look-up returns only a small amount of results as it is usually the case for `(s,?,?)`, `(s,?,o)`, `(s,p,o)` queries. However, the more query results that need

[7] We first list the LUBM datasets in increasingly order of triples, and use name abbreviations for LUBM (L), Census (C), Jamendo (J), and AEMET (A).

[8] Results are comparable for all datasets.

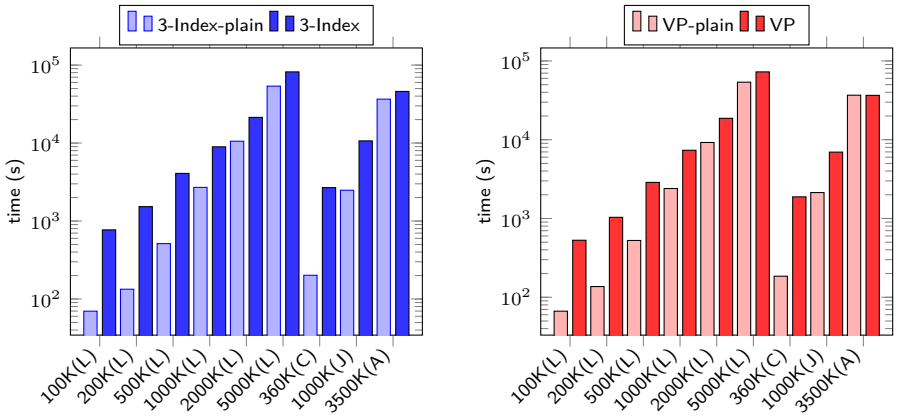

Fig. 4: Time for loading (encrypting+indexing) the entire dataset for `3-Index` and `VP`. We only report indexing time for the non-secure counterparts `3-Index-plain` and `VP-plain`.

to be returned the longer the decryption takes. At this point we also want to stress that due to the nature of our approach, each result triple can be returned as soon as its decryption has finished. This is in line with the incremental nature of the Triple Pattern Fragment [31] approach, which paginates the query results (typically including 100 results per page), allowing users to ask for further pages if required. For example, decrypting Jamendo entirely took about 2256s for `VP` and 2808s for `3-Index`, leading to respective triple decryption rates of 465 triples/s and 374 triples/s in a cold scenario, which already fulfils the performance requirements to feed several Triple Pattern Fragments per second.

Scalability. As mentioned in Section 4, our approach allows for parallel encryption/decryption of triples, thus scales with the system's supported level of parallelisation/number of available cores (e.g., encrypting and indexing (`3-Index`) 10000 LUBM triples takes about 76s with 16 available cores, 133s with 8, 262s with 4, and 497s with 2 available cores).

Our experiments have shown that (i) the performance of our indexing strategy is not affected by the encryption, hence, is as effective on encrypted data as it is on non-encrypted data, and (ii) the decryption of individual triples is a fast process which can be utilised in our Linked Data scenario, especially under the umbrella of the Linked Data Fragments framework.

6 Conclusion

To date Linked Data publishers have mainly focused on exposing and linking open data, however there is also a need to securely store, exchange, and query also sensitive data alongside (i.e., closed data). Both access control and encryption mechanisms are needed to protect such data from unauthorised access, security breaches, and potentially untrusted service providers. Herein, we presented a

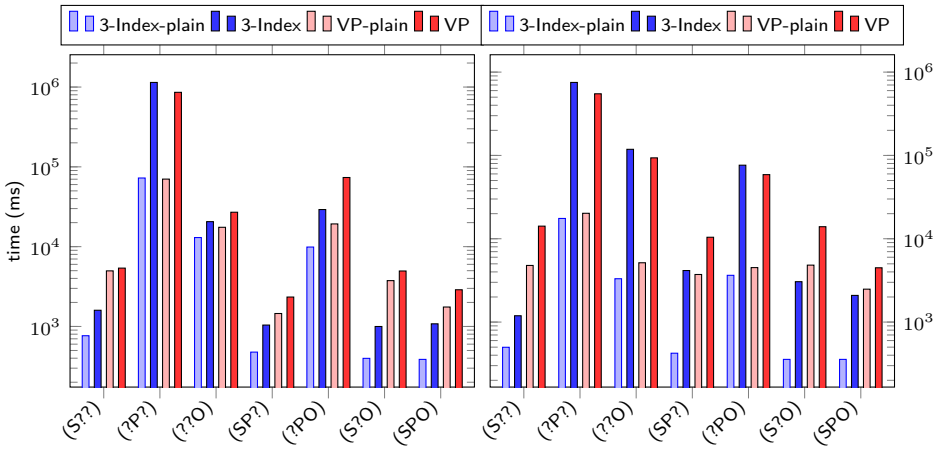

Fig. 5: Cold query times of LUBM with 5M triples (LHS) and Jamendo (RHS) for `3-Index`, `VP`, and their non-secure counterparts in ms (logarithmic y-axis).

mechanism to provide secure and fine-grained encryption of RDF datasets. First, we proposed a practical realisation of a functional encryption scheme, which allows data providers to generate query keys based on (triple-)patterns, whereby one decryption key can decrypt all triples that match its associated triple pattern. As such, our approach operates on a very fine level of granularity (i.e., triple level), which provides a high degree of flexibility and enables controlled access to encrypted RDF data. In existing literature, enforcing access control at the level of single statements or tuples is generally referred to as fine-grained access control (cf. [23]). Then, we presented two indexing strategies (implemented using MapDB) to enhance query performance, the main scalability bottleneck when it comes to serving user requests.

Our empirical evaluation shows that both indexing strategies on encrypted RDF data report reasonable loading and query performance overheads with respect to traditional, non-encrypted data retrieval. Our results also indicate that the approach is relatively slow for batch decryption, but this can be counteracted by the fact that it is suitable for serving incremental results, hence it is particularly suitable for Linked Data Fragments.

In future work, we plan to inspect different indexing strategies in order to optimise the loading time and query performance of large queries. We also consider extending our proposal to cater for named graphs, that is, encrypting quads instead of triples and generating keys based on quad patterns. Finally, we aim to integrate the proposed secure RDF store with a "policy" tier by employing Attribute-based Access Control (ABAC), which will manage the access/revocation to the query keys and serve as fully fledged security framework for Linked Data.

References

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proc. of Very Large Data Bases*, pages 411–422, 2007.

[2] M. Abdalla, F. Bourse, A. D. Caro, and D. Pointcheval. Simple functional encryption schemes for inner products. In *Proc. of the 18th International Conference on Practice and Theory in Public-Key Cryptography*, pages 733–751, 2015.

[3] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world sparql queries. *arXiv preprint arXiv:1103.5043*, 2011.

[4] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Proc. of Advances in Cryptology*, pages 353–373. 2013.

[5] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. *IACR Cryptology ePrint Archive*, 2014:853, 2014.

[6] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Proc. of Applied Cryptography and Network Security*, pages 442–455, 2005.

[7] M. Chase and E. Shen. Pattern matching encryption. *IACR Cryptology ePrint Archive*, 2014:638, 2014.

[8] P. Cudré-Mauroux, I. Enchev, S. Fundatureanu, P. Groth, A. Haque, A. Harth, F. L. Keppmann, D. Miranker, J. F. Sequeda, and M. Wylot. NoSQL databases for RDF: an empirical evaluation. In *Proc. of International Semantic Web Conference*, pages 310–325, 2013.

[9] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proc. of Computer and communications security*, pages 79–88, 2006.

[10] P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, and M. J. Wheelhouse. A simple abstraction for complex concurrent indexes. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications*, pages 845–864, 2011.

[11] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[12] A. De Caro and V. Iovino. jpbc: Java pairing based cryptography. In *Proc. of IEEE Symposium on Computers and Communications*, pages 850–855, 2011.

[13] S. D. C. di Vimercati, S. Foresti, G. Livraga, and P. Samarati. Practical techniques building on encryption for protecting and managing data in the cloud. In *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, pages 205–239, 2016.

[14] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF representation for publication and exchange (HDT). *J. Web Sem.*, 19:22–41, 2013.

[15] C. Gentry et al. Fully homomorphic encryption using ideal lattices. In *Proc. of ACM Symposium on Theory of Computing*, volume 9, pages 169–178, 2009.
[16] S. Gerbracht. Possibilities to Encrypt an RDF-Graph. In *Proc. of Information and Communication Technologies: From Theory to Applications*, pages 1–6, 2008.
[17] M. Giereth. On Partial Encryption of RDF-Graphs. In *Proc. of International Semantic Web Conference*, volume 3729, pages 308–322, 2005.
[18] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics*, 3(2):158–182, 2005.
[19] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000.
[20] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial cryptography and data security*, pages 258–274. 2013.
[21] A. Kasten, A. Scherp, F. Armknecht, and M. Krause. Towards search on encrypted graph data. In *Proc. of the International Conference on Society, Privacy and the Semantic Web-Policy and Technology*, pages 46–57, 2013.
[22] J. Katz, A. Sahai, and B. Waters. Predicate Encryption Supporting Disjunctions, Polynomial Equations, and Inner Products. *J. Cryptology*, 26(2): 191–224, 2013.
[23] S. Kirrane, A. Mileo, and S. Decker. Access control and the resource description framework: A survey. *Semantic Web*, 8(2):311–352, 2017. doi: 10.3233/SW-160236. URL http://dx.doi.org/10.3233/SW-160236.
[24] K. Kurosawa and L. T. Phong. Kurosawa-desmedt key encapsulation mechanism, revisited. *IACR Cryptology ePrint Archive*, 2013:765, 2013.
[25] G. Ladwig and A. Harth. CumulusRDF: linked data management on nested key-value stores. In *Proc. of Scalable Semantic Web Knowledge Base Systems*, page 30, 2011.
[26] A. B. Lewko, T. Okamoto, A. Sahai, K. Takashima, and B. Waters. Fully secure functional encryption: Attribute-based encryption and (hierarchical) inner product encryption. In *Advances in Cryptology*, pages 62–91, 2010.
[27] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology*, pages 223–238, 1999.
[28] R. Popa, N. Zeldovich, and H. Balakrishnan. Cryptdb: A practical encrypted relational dbms. Technical report, MIT-CSAIL-TR-2011-005, 2011.
[29] Y. Sagiv. Concurrent Operations on B*-Trees with Overtaking. *J. Comput. Syst. Sci.*, 33(2):275–296, 1986.
[30] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *Proc. of Network and Distributed System Security*, volume 14, pages 23–26, 2014.
[31] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert. Triple Pattern Fragments: a low-cost knowledge graph interface for the Web. *Journal of Web Semantics*, 37–38:184–206, Mar. 2016.
[32] E. Zheleva and L. Getoor. To join or not to join: the illusion of privacy in social networks with mixed public and private user profiles. In *Proc. of World Wide Web*, pages 531–540, 2009.