Rheinisch-Westfälische Technische Hochschule Aachen
Informatik 5, Information Systems
Prof. Dr. Stefan Decker

BACHELOR THESIS PROPOSAL

Fully homomorphic encryption RDF-Graphs

Yoann Maurice Kehler

April 13, 2018

Advisor	Benjamin Heitmann, Ph.D.
1st Supervisor	Prof. Dr. Stefan Decker
2nd Supervisor	Prof. Dr. Mustermann

Contents

1 Introduction

1.1 Motivation

Migrating services to the cloud, expands the boundaries of a customers network over to the cloud provider. While this might cause no issue for some services, those dealing with sensitive data have to consider new privacy threats. Not only do security flaws in the cloud providers network threaten the entrusted data privacy, but so does the cloud provider itself. Sensitive data stored unencrypted on a foreign system is accessible for theft and abuse. Imagine a number of hospitals storing their patients sensitive medical data in the cloud, in order to share knowledge across their facilities, and maybe to gain further insights through big-data techniques. Obviously traditional approaches with symmetric and public key encryption do not work, since, they all require the cloud provider to own a set of keys, and by doing so, the customer gives it full access to the sensitive data. Luckily, homomorphic encryption (HE) attempts to enable the evaluation of functions on encrypted data. The data processor does not have to be trusted any longer. Although, numerous libraries implement basic fully homomorphic encryption (FHE) schemes, a gap between the high level requirements of a real-world application and the low-level libraries exists. They miss by far the functions we are used to from high level programming languages and even simple tasks and more complex data types turn out to be non-trivial.

By nature, calculations delegated to the cloud are expensive, otherwise they could run on a local machine in the first place. Also they tend to deal with a lot of data accessed by many user. For example hospitals might store their patient data in a remote data base, which gives all their employees easy access and allows information gathering through a complex query language.

1.2 Thesis Goal

The ultimate goal would be to take a complex data type from a common usecase, such as Resource Description Framework (RDF) graphs, and, based on a FHE library, implement a working encrypted prototype. It should integrate with a common API from an existing non-ecrypted RDF graph an have a useful set of operations, while, at the same time, being efficient enough for every day use.

Due to the limited scope of a bachelor thesis the thesis will most probably tackle only the lower layers in detail and then draw an outline towards a fully functional encrypted RDF graph.

On the way, the capabilities and limitations of modern FHE libraries should be discerned, as well as their theoretic fundamentals explained.

1.3 Outline

In a first step, this proposal will introduce the fundamentals of FHE. Furthermore, it will present the basic concepts of linked data and their most important data structure - the RDF graph, alongside with some related works in chapter 2.

In a second step, it will derive the requirements of an encrypted RDF graph in chapter 3 and draw a layered conceptual approach to the problem in chapter 4.1, with some additional considerations regarding the implementation.

Eventually, the proposal will explain how I intent to evaluate my work and, at last, is giving an outline of my time schedule for coming four months.

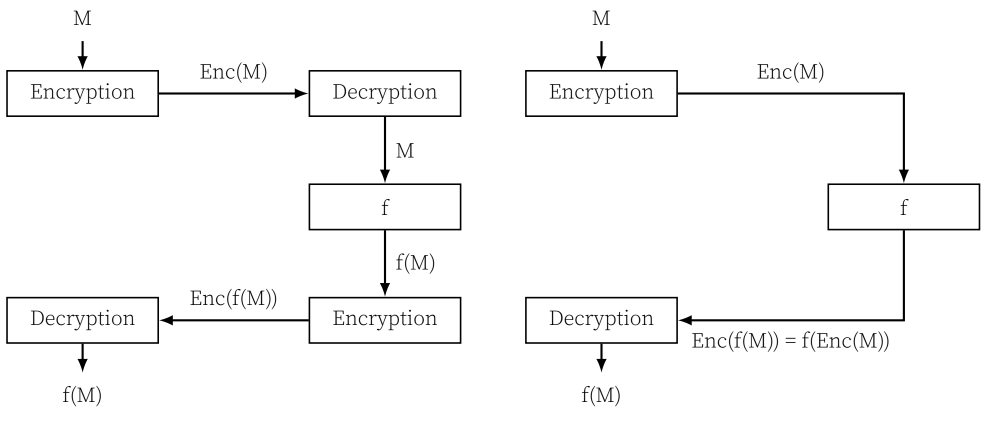
2 Related Work

2.1 Fully Homomorphic Encryption

Initial idea The fully homomorphic encryption (FHE) is an old idea of cryptography. The goals of a fully homomorphic encryption scheme were formalized first by Rivest et al. in 1978[13]. It should allow the computation of any arbitrary function on encrypted data, without the need to decrypt it first. An easy example for homomorphic encryption is textbook RSA which allows any function over the algebraic system $(\mathbb{Z}_p, \cdot_p, \equiv_p)$; the integers modulus a prime p with a multiplication $\cdot_p$ and an equality $\equiv_p$.

$$E(x) \cdot_p E(y) = x^e \cdot_p y^e = (x \cdot_p y)^e = E(x \cdot_p y)$$

In a two party set-up FHE avoids decryption by the computing party as shown in figure 2.1. The message is therefore unknown to the foreign party and safe from attackers on the their side of the communication channel.



(a) non-homomorphic encryption (b) homomorphic encryption

Figure 2.1: Remote computation with non-homomorphic encryption and homomorphic encryption.

Theoretic breakthrough The first truly fully homomorphic encryption scheme was invented by Gentry only in 2009 [10] thirty years after the initial proposition. His construction was done in three main steps. First he put forward a so-called somewhat homomorophic encryption (SHE), which is fully homomorphic for arithmetic circuits up to a certain depth. His proposed SHE scheme was based on ideal lattices. In a second step, the decryption algorithm is "squashed" to fit into the circuit restrictions of the encryption scheme it self. In order to get there Gentry was forced to make additional hardness-assumptions, besides the hardness assumption of the underlying theoretical problem of the scheme. As it turned out later, this assumption can be dropped for some SHE schemes. [5] The third and most crucial step in his construction was the so-called bootstrapping. Since the decryption could now be computed homomorphically inside an encrypted message, overly complex functions could be broken down in multiple steps. Whenever the restriction of the SHE is attained, the partially computed encrypted result can be re-encrypted and the inner encryption can be decrypted in a homomorphic manner, leaving extra room for the next step.

More formal, for a function f, which can be broken down into subfunctions

$$f = f_n \circ \cdots \circ f_1$$

a bootstrapping step schematically looks like the following:

$$E_1(M) \xrightarrow{f_1} E_1(f_1(M)) \xrightarrow{E_2} E_2(E_1(f_1(M))) \xrightarrow{D_1} E_2(f_1(M)) \xrightarrow{f_2} E_2(f_2 \circ f_1(M))$$

Obviously, the SHE scheme is transformed into a FHE scheme, as arbitrary complex functions are computable on an encrypted cipher-text. However, the scheme was not perfect. Apart from the mentioned hardness assumption, the encryption scheme hat cumbersomely large cipher-texts and performance was bad overall.

Modern FHE schemes The bootstrapping technique was applied to numerous other SHE schemes in order to achieve a full homomorphism.

Brakerski and Vaikuntanathan based their SHE scheme on the Learning With Errors (LWE) and managed to eliminate the squashing step and it's attached hardness assumption entirely in doing so.[5] Later, they even disposed of the costly bootstrapping step resulting in the Brakerski-Gentry-Vaikuntanathan (BGV) encryption scheme. Instead, they leveraged the properties of LWE and introduced a noise in the cipher-text, which if it gets to big, makes the message impossible to decrpyt. However, if necessary, this noise can still be reduced to a constant level, by doing a bootstrapping step. [4]

Fan and Vercauteren ported this scheme to the Ring-Learning With Errors (Ring-LWE) problem presented by Regev. [12][8] Additionally, The Full-RNS variant of this scheme makes some more low level improvements, without touching it's fundamental way of operation.[1]

Homomorphic encryption libraries HELib[11] is based on the BGV encryption scheme, with numerous performance improvements. It provides very low-level routines and, according to them, is mostly aimed at researchers. The Simple Encrypted Arithmetic Library (SEAL)[6] is aimed at practicality and easy use. It is based on the FullRNS variant of the Fan-Vercauteren (FV) scheme. Another library worthy to mention is the Fast Fully Homomorphic Encryption Library over the Torus (TFHE), which also based on the BGV scheme.

2.2 Linked Data

The World Wide Web (WWW) was invented by Tim Berners Lee as an interconnection of documents formatted in Hypertext Modeling Language (HTML), which can all be accessed by a Unified Resource Locator (URL) through Hypertext Transfere Protocol (HTTP).[3] It is, with no doubt, one cornerstones of the Internet how we know it today. The possibility to reference documents throughout the Internet changed the way how informations are shared in a fundamental way. The Web browser has become a central if not the most used application on ones computer.

Before the invention of Linked data, data used to be published in raw formats and it had lost it's semantics. In order to give it some meaning, additional informations such as a human readable descriptions or visualizations were required. Linked data however is an approach, which applies the same principles – Unified Identifiers, Data Model and Protocols – to data itself, yielding a graph like (”linked”) data structure, which will be described in the following sections in a more formal way.

For now lets just say, every element is referenced/named by a Unified Resource Identifier (URI). Since the entire data is separately stored on different servers, they need to store it in a standardised fashion, defined by the World Wide Web Consortium (W3C), with a common data model, the Resource Description Framework (RDF). The semantics are defined with the Web Ontology Language (OWL) and Data queries are usually done in SPARQL Protocol And RDF Query Language (SPARQL).

Uniform Resource Identifier

URIs are defined by the W3C in RFC3986.[2] URLs are URIs, thus, URI often resemble an URL. A URI is a Unicode strings without white spaces, consisting of five parts. A scheme, authority, path, query and fragment, of which some are optional. Take a look at the reference, for further information. Two example URIs are displayed in Figure 2.2.

RDF Graphs

The W3C defines RDF Graphs as follows[7]:

```
foo://example.com:8042/over/there?name=ferret#nose
\_/   \______________/\_________/ \_________/ \__/
 |           |            |            |        |
scheme     authority       path        query   fragment
 |   _____________________|__
/ \ /                        \
urn:example:animal:ferret:nose
```

Figure 2.2: An example URI from RFC3986[2]

Definition 2.2.1 (RDF Triples)
An RDF triple consists of three components:
***the subject**, which is an IRI or a blank node*
***the predicate**, which is an IRI*
***the object**, which is an IRI, a literal or a blank node*
An RDF triple is conventionally written in the order subject, predicate, object.

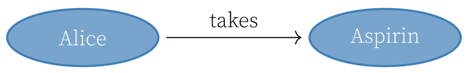


Figure 2.3: Representation of a Triple

Definition 2.2.2 (RDF Graph)
An RDF graph is a set of RDF triples. [...]
The set of nodes of an RDF graph is the set of subjects and objects of triples in the graph. It is possible for a predicate IRI to also occur as a node in the same graph.

Blank nodes are only localy scoped identifiers and not well defined in the standard. It is enough to know, that if necessary blank nodes can be replaced by URLs, leading to a blank node free graph.

The set of triples below is equivalent to the graph visualized in figure 2.4.

$$
\begin{array}{l}
Triples = \{ \\
\qquad (Alice, takes, Aspirin) \\
\qquad (Bob, takes, Aspirin) \\
\qquad (Bob, takes, Paracetamol) \\
\qquad (Charlie, takes, Ibuprofen) \\
\}
\end{array}
$$

Of course RDF graphs can contain a lot more triples and become far more complex. The prominent RDF graph ”DBpedia” contains roughly 4.5 Million things at the moment (April 2018).

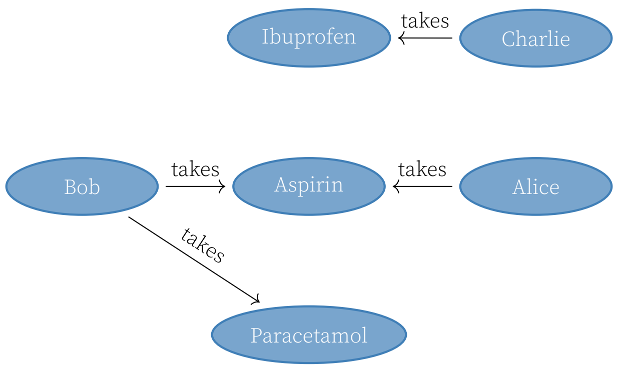


Figure 2.4: A small RDF graph, containing medical data

Search queries

Now given a RDF graph, a user can search informations inside the graph by issuing a triple containing a search pattern. That is, the terms can be replaced by wild-card place holders, in which case all triples satisfying the remaining terms form the resulting set.

In our example a client can retrieve „All patients taking Aspirin“with the query

$$Query = (*, takes, Aspirin).$$

The result is the set

$$Result = \{(Bob, takes, Aspirin), (Alice, takes, Aspirin)\}.$$

The result can be considered as a RDF graph again. More complex queries are possible, by taking the union of resulting sets or applying multiple queries one after another.

3 Use Case and Requirements

Let's take a step back and consider the hospital scenario from the introduction again. It implies a single server running an encrypted Resource Description Framework (RDF) graph, which is accessed by multiple clients. The server, does not know the content of the graph, as it is encrypted and the server does not hold any secret information regarding this encryption.

For simplicity we will consider only a single client scenario, since secret sharing, authorization techniques and multi user interfaces are already solved by other domains of computer science.

Beforehand the encryption is set up and an empty graph is created on the server. The secret information for the encryption is stored safe by the client, in order to grant access on the encrypted graph, in the future. The client can then fill the graph with informations from remote and search the graph, through an elaborate query language and RDF library.

3.1 Requirements of an encrypted RDF-Graph

The requirements for an encrypted RDF Graph can be looked at from three standpoints.

Functional requirements

The Jena-API provides an interface for their RDF-Graph. It demands four functions from which all the other functions are derived:

`void performAdd(Triple)` Adds a given triple to the graph.

`void performDelete(Triple)` Deletes a given triple from the graph.

`Iterator<Triple> graphBaseFind(Triple)` Returns an Iterator to a lists of triples from the graph, which match the searchpattern given as triple.

The homomorphic encrypted graph should provide those functions, whereas the input is encrypted or not. For the user, the encryption should be as transparent as possible. That is, once the encryption and connection to the remote server are set up, the graph should behave as if, it was not encrypted and is running on the local machine.

The graph must support arbitrary many search operations, without additional interaction with the client. In contrast, the noise will probably grow with every add- and delete-operation. A predefined number of those should be feasible before performing a bootstrapping-step, which requires more complex interaction with a client.

Security requirements

At no point in time, the server should gain access to the sensitive data stored inside the graph. That implies, the server holds no secret information regarding the encryption.

Also the server should not be able to derive any sensible information from the graph, by analysing the search-queries, traffic to and from the server, as well as the encrypted graph itself.

The size of the graph, number of triples, are known to the server. If necessary, more structural information about the graph can be given to server in order to make an encrypted Graph even possible.

Performace requirements

In order to stay usable, a working encrypted RDF graph should perform an action in no longer then a couple of minutes. Still, performance is not the main goal of this thesis and the result might turn out to be to slow for actual use.

The encrypted RDF graph's storage size should be limited by a constant factor relative to the original size of the data and the operations should run with a normal sized memory (roughly 16GB) on a modern Processor (Intel i7).

4 Approach

4.1 Conceptual Approach

Implementing a high level application, like a RDF graph on top of an fully homomorphic encryption (FHE) libraries requires more low level functionalities to be programmed beforehand.

An unencrypted graph for instance builds upon an RDF-Library like Apache Jena [9] with a query language, namely SPARQL Protocol And RDF Query Language (SPARQL). Those however rely on a simpler data structure, such as a triple store, which stores the content of the graph in a simplified manner. This in turn, uses other complex data structures - Lists, Vectors and so on, which at some point use only the primitive data-types provided by the programming language.

Similarly, all those layers must be implemented with homomorphic encryption, before actually building an encrypted RDF graph. The layered architecture is visualized in figure 4.1. The thesis, will work it's way from the bottom layer to the top layer.

4.2 Bridging the gap

Since even floating-point numbers, strings and characters, which are considered as primitive data-types in most high-level programming languages do not exist in the common FHE libraries, they must be tackled first.

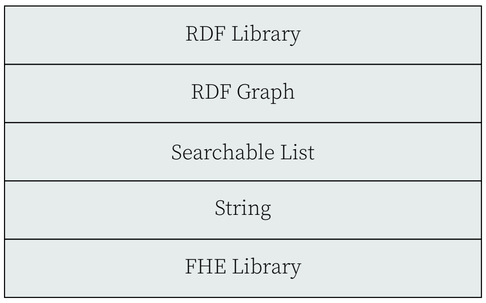


Figure 4.1: The conceptual layers between the high level RDF Library and the low-level FHE library

String Luckily, only the string type seems relevant for the implementation of an encrypted RDF graph, since the Unified Resource Identifiers (URIs) for the terms will be stored as such and they are the only sensitive data in the graph. Nevertheless, the string type is complex in it self. So as to make it fully functional for our application, it must support arbitrary long content, comparison (with a wild-card symbol), and Unicode.

Triple As seen in section 2.2 an RDF-triple is simply a subject, a predicate and an object, which are all referenced by URIs, which in turn are simply strings. Once those exist, the triple follows naturally.

List Since the graph is nothing else then a set of triples, those triples can be stored in list, and searches on this list are performed (not regarding the performance) linearly, comparing every single item with the given search query.

4.3 Implementation

For the project I will mostly rely on the Simple Encrypted Arithmetic Library (SEAL)[6]. It is programmed in C/C++ and compiles without any dependencies. It's application programming interface (API) is clean and easy to use. Accordingly, the most cryptographic work of the project will happen in C++. The implementation of the graph, however, will be done in Java, since, the target Interface `GraphBase` is provided by Apache Jena. Somewhere in between the transition from C++ to Java will happen through Java's Native Interface (JNI).

5 Evaluation

The implemented encrypted graph, will be evaluated regarding requirements stated in section 3.1.

Functional Requirements will be evaluated with the help of test cases on the several architectural layers, in order to guarantee the correctness of the implementation.

Security Requirements will be derived from the properties of the underlying FHE library.

Performance Requirements will be evaluated empirically on my personal Computer. The Encrypted Triple will be tested on some examples, and the the runtime of the encryption, decryption and comparision as well as the ciphertext size and ram usage. For an encrypted graph, the core functions (add, delete, find) will run on several reasonable sized graphs. Again the runtime of those operations (dependently on the graph's size) and the RAM Usage are of interest. Given the time, those measurements will be compared to theoretical approximations of the problem.

It might be necessary to trade in some security requirements for higher performance. In this case, the different sollutions will be compared to one another and proofs about the remaining security will be given.

6 Timeline and Project Plan

The bachelor thesis is meant to be written in a time period of no longer than four months. The project will be devised in subtasks as seen in the time schedule in table 6.1.

Proposal Presentation In the first week, this document will be summed up in a twenty minutes long talk, with intent to present my project to other interested students in the field and gather extra input from them.

Practical Research Then after getting having green light, the first step is to dive into actual problem solving with FHE libraries and learn their basic functionalities.

Theoretical Research and Write Ups Most probably, more background research must be done in addition to this proposal document. Alongside, core proofs and concepts will be written down in Latex to facilitate the later writing phase.

Implemenation The most time will be spent on Implementation of the homomorphic encrypted data-types, presented in chapter 4.1.

Evaluation The results will then be evaluated, according to the design goals and techniques described in chapter 5.

In week 9 the minimum goal of the thesis, namely a homomorphically encrypted string, should be attained, with a prototype of the written thesis. In a second iteration, this solution should then be improved and if possible extended towards an encrypted RDF graph. The final thesis should stand by the beginning of August.

Table 6.1: Time Schedule

	KW	**Week**	**Activity**
April	16	1	Proposal Presentation
	17	2	Practical Research
May	18	3	Theoretical Research and Write Ups
	19	4	Implementation 1
	20	5	Implementation 1
	21	6	Implementation 1
June	22	7	Evaluation
	23	8	Writing Thesis
	24	9	Writing Thesis
	25	10	Research and Write Ups
	26	11	Implementation 2
July	27	12	Implementation 2
	28	13	Evaluation
	29	14	Writing Thesis
	30	15	Writing Thesis
August	31	16	Writing Thesis
	32	17	Buffer
	33	18	Buffer

Abbreviations

API	application programming interface
BGV	Brakerski-Gentry-Vaikuntanathan
FHE	fully homomorphic encryption
FV	Fan-Vercauteren
HE	homomorphic encryption
HTML	Hypertext Modeling Language
HTTP	Hypertext Transfere Protocol
LWE	Learning With Errors
OWL	Web Ontology Language
RDF	Resource Description Framework
Ring-LWE	Ring-Learning With Errors
SEAL	Simple Encrypted Arithmetic Library
SHE	somewhat homomorophic encryption
SPARQL	SPARQL Protocol And RDF Query Language
TFHE	Fast Fully Homomorphic Encryption Library over the Torus
URI	Unified Resource Identifier
URL	Unified Resource Locator
W3C	World Wide Web Consortium
WWW	World Wide Web

References

[1] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca. “A full RNS variant of FV like somewhat homomorphic encryption schemes”. In: *International Conference on Selected Areas in Cryptography*. Springer. 2016, pp. 423–442.

[2] T. Berners-Lee, R. T. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. STD 66. http://www.rfc-editor.org/rfc/rfc3986.txt. RFC Editor, Jan. 2005.

[3] C. Bizer, T. Heath, and T. Berners-Lee. “Linked data-the story so far”. In: *Semantic services, interoperability and web applications: emerging concepts* (2009), pp. 205–227.

[4] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. *Fully Homomorphic Encryption without Bootstrapping*. Cryptology ePrint Archive, Report 2011/277. https://eprint.iacr.org/2011/277. 2011.

[5] Z. Brakerski and V. Vaikuntanathan. “Efficient fully homomorphic encryption from (standard) LWE”. In: *SIAM Journal on Computing* 43.2 (2014), pp. 831–871.

[6] H. Chen, K. Han, Z. Huang, A. Jalali, and K. Laine. *Simple Encrypted Arithmetic Library*. Version 2.3.0. Feb. 2018.

[7] R. Cyganiak, D. Wood, and M. Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/. Feb. 2014.

[8] J. Fan and F. Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2012/144. https://eprint.iacr.org/2012/144. 2012.

[9] T. A. S. Foundation. *Apache Jena*. Version 3.7.0. Feb. 2018.

[10] C. Gentry. “Fully Homomorphic Encryption Using Ideal Lattices”. In: *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*. STOC ’09. Bethesda, MD, USA: ACM, 2009, pp. 169–178.

[11] S. Halevi. *HElib*. Version 1.3. Nov. 2017.

[12] O. Regev. “The learning with errors problem”. In: *Invited survey in CCC* (2010), p. 15.

[13] R. L. Rivest, L. Adleman, and M. L. Dertouzos. “On Data Banks and Privacy Homomorphisms”. In: *Foundations of secure computation* (1978), pp. 169–177.