CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

On Preserving Privacy

Queries to Cloud Stored Homomorphically Encrypted Documents

Master's thesis in Computer science and engineering

Tarek Chorfi, Pavlos Stampoulis

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

On Preserving Privacy

Queries to Cloud Stored Homomorphically Encrypted Documents

Tarek Chorfi

Pavlos Stampoulis

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

On Preserving Privacy
Queries to Cloud Stored Homomorphically Encrypted Documents
Tarek Chorfi
Pavlos Stampoulis

Supervisor: Rhouma Rhouma, Computer Science and Engineering
Examiner: Ahmed Ali-Eldin Hassan, Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

On Preserving Privacy
Queries to Cloud Stored Homomorphically Encrypted Documents
Tarek Chorfi
Pavlos Stampoulis
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis explores homomorphic encryption on queries to cloud stored documents. Moreover, the aim of this thesis is to explore the use of semantic search on encrypted word embeddings with end-to-end privacy. A concrete implementation of a secure semantic search application that stores documents in a database which allows for efficient retrieval (using Locality Sensitive Hashing) and computation of embedding similarities is presented. Experiments were conducted to benchmark the performance of homomorphic operations on encrypted data. CKKS was the homomorphic encryption scheme used in these experiments, because CKKS works with vectors of real numbers, which is what word embeddings are. The experiments focused on how much of the operations can be offloaded to the server and also the accuracy between decrypted ciphertexts and plaintexts after computations. Our results show that achieving high accuracy between decrypted ciphertexts and plaintexts does not decrease performance, however it does limit the level of security depending on the set parameters. We concluded that homomorphic encryption is feasible for our specific use-case and could potentially allow almost 300 000 similarity computations per second given a server cluster of 8 hosts each having an Nvidia 4090 GPU.

Keywords: Homomorphic encryption, Word embeddings, CKKS, Semantic search, Locality Sensitive Hashing

Acknowledgements

We would primarily like to thank our families for their support during this project. We would also like thank our supervisor Rhouma Rhouma for guiding us throughout this thesis, and our examiner Ahmed Ali-Eldin Hassan for the support and helpful comments.

Tarek Chorfi, Pavlos Stampoulis, Gothenburg, 2025-07-16

Contents

List of Figures

List of Tables

1

Introduction

Homomorphic encryption (HE) is a field of cryptography where encryption schemes allow for computations to be performed on encrypted data without the need to decrypt ciphertexts. This allows anonymous computations to be performed. This can be applicable in many known sectors, some of which are cloud computing, financial institutions, voting and healthcare [1]. Homomorphic encryption schemes are promising, however the current drawbacks are factors such as high latency and computational overhead [2]. Additionally, as quantum computing continues to evolve, it is crucial for cryptography and data encryption to follow suit – or, ideally, get ahead. After all, with recent breakthroughs in quantum computing, the need for post-quantum secure cryptography has never been more urgent. For example, Microsoft's road to a million qubits with its Topological Core has accelerated development from decades to just years [3]. Since HE is post-quantum secure, it could act as a key candidate for future privacy preserving services.

1.1 Purpose

The purpose of this thesis is evaluate the feasibility and privacy benefits of using homomorphic encryption in vector similarity search. Vector similarity search is a mathematical way of comparing vectors to calculate their similarity, often being a measure of angular similarity or the distance between two vectors. Word embeddings are encodings of natural language that are n-dimensional vectors that contain the semantic meaning of the input. Because word embeddings encode the semantic meaning of the input, it is possible to perform semantic search (a term often used for vector similarity search where the semantics of the input are encoded in the vectors) on word embeddings in the vector domain which is consistent with the plaintext domain. We aim to present benchmarks of HE used within word embeddings to measure accuracy, performance and scalability for vector similarity search, whilst also assessing its privacy- and security level.

1.2 Envisioned use-case

We want to explore a solution for a secure embedding database (i.e. secure cloud storage). We believe that it could be used within AI augmented questioning and answering when dealing with sensitive source data. Retrieval Augmented Generation

(RAG) is a method in which one can leverage Large Language Models to answer questions given input contexts. There has not been significant work in the field that combines vector databases with homomorphic encryption. Thus we intend to explore a solution for a secure embedding storage that allows for private retrieval by a third-party client.

In the complete use-case, a user should be able to upload documents without divulging any information about their data, this means that the documents stored should be encrypted. The user should also be able to perform semantic search on the stored data without divulging any information about the query. This means that the query will also be encrypted. By leveraging homomorphic encryption we can do mathematical operations on ciphertexts. Thus no information about the retrieved data or query is leaked, because both are unknown to the server.

The user can then with the help of an LLM (preferably hosted locally for privacy) pass the query along with the decrypted retrieved data to allow for easy question and answering functionality. This will allow the user to get an answer to their question, given their input data, and help speed up efficiency when searching through sensitive large-scale data.

1.3 Research questions

We will focus on presenting HE's applicability for one use-case, similarity search in encrypted word embeddings. The research questions we want to answer are:

Research question 1: How can privacy be preserved with HE?

Research question 2: What are the limitations of HE in our implementation?

Research question 3: Is it viable to use HE in the real-world for the previously discussed use-case?

1.4 Scope and constraints

Security is a big part of cryptography, if the scheme is not secure then it should not be used. Our security assessment of our HE implementation will not be about the mathematical theory behind it. There already exist extensive papers on the hardness assumption and proofs of Learning with Errors (LWE) and Ring-Learning with Errors (RLWE). LWE and RLWE are mathematical problems where noise (error) is added to the secret information [4]; further explanation is provided in Section 2.1.1 and Section 2.1.2. One notable work studying the theoretical aspects of HE is the work by Lyubashevsky et al.[5], where they provide rigorous complexity-theoretic reductions of the LWE-problem which the HE schemes are based on. Therefore, it will not be within the scope of this project to perform a theoretical and mathematical security analysis of HE schemes. However, we will discuss the security constraints and trade-offs from the different experiments conducted in this thesis.

2

Theory and Background

This chapter will provide a sufficient understanding of the theory and concepts on which the implementation and experiments are based on. Grasping the core idea of the theory and how it works is more important than explaining mathematical definitions and proofs. Additionally, this chapter provides a technical understanding of the operations used, in order to understand the report's result and discussion. The reader is expected to have a basic grasp of group theory and rings, as well as an understanding of linear algebra. Additionally, some familiarity with Natural Language Processing would be beneficial, however not necessary.

2.1 Homomorphic Encryption

Homomorphic encryption allows for operations to be performed on the ciphertext, known as a malleable ciphertext. The decrypted message will be consistent with the operations performed on the message whilst it was encrypted. Furthermore, the components needed for a general HE scheme are presented in Definition 1.

Definition 1 (Homomorphic Encryption (HE) Scheme [2]) *A homomorphic encryption scheme using public-key encryption is set to contain four components:*

1. *KeyGen: Takes as input the security parameter, often denoted as λ, then generates a public-key and a secret-key.*
2. *Encrypt: Uses the public-key (or secret-key) to map the plaintext into a ciphertext.*
3. *Eval: Some function $f()$ takes i amount of ciphertexts, where $i \in \mathbb{N}$, and applies some operation on $\{c_0, c_1, \ldots, c_i\}$, outputting a new ciphertext corresponding to the result of the same operation on the underlying plaintexts.*
4. *Decrypt: Uses the secret-key to map the ciphertext back into a plaintext.*

In summary, the homomorphic property guarantees that evaluating a function on ciphertexts and then decrypting the result yields the same outcome as applying the function directly to the corresponding plaintexts. Moreover, in HE there exist schemes which are designed for either public-key or secret-key encryption. Depending on the intended use of the HE scheme, both public-key and secret-key encryption offer distinct advantages. Secret-key encryption is preferable in settings where data generation and computation are handled internally, as it ensures that only one party

can encrypt and decrypt the data. Public-key encryption, on the other hand, enables multiple external parties to encrypt data using a shared public key. This is particularly useful for scenarios such as offloading computations to untrusted servers or storing encrypted data in the cloud while preserving privacy.

2.1.1 Lattices and Learning with Errors

Lattices and learning with errors are the fundamental mathematics and theoretics used to create homomorphic encryption schemes. This section aims to clarify these fundamentals in order to provide a deeper insight into how HE schemes actually function.

Definition 2 (Lattices) [6] *A lattice $\mathcal{L}$ is a set of infinite points in the real coordinate space $\mathbb{R}^n$, $\mathcal{L} \in \mathbb{R}^n$, more accurately $\mathcal{L}$ is an additive discrete sub-group of $\mathbb{R}^n$. A lattice has a vector basis in $\mathbb{R}^n$, respectively a vector basis has any linear combination with integer coefficients to form a lattice.*

As shown in Figure 2.1, going from left to right is how lattices are created. Column A represents the basis vectors. Column B represents the paths of the vectors depending on their coefficients. Column C is showing the clear lattices created by the basis vectors. Row number one is a hexagonal lattice pattern. Row number two is a 45° rotated large square pattern. Row number three is a normal square pattern.

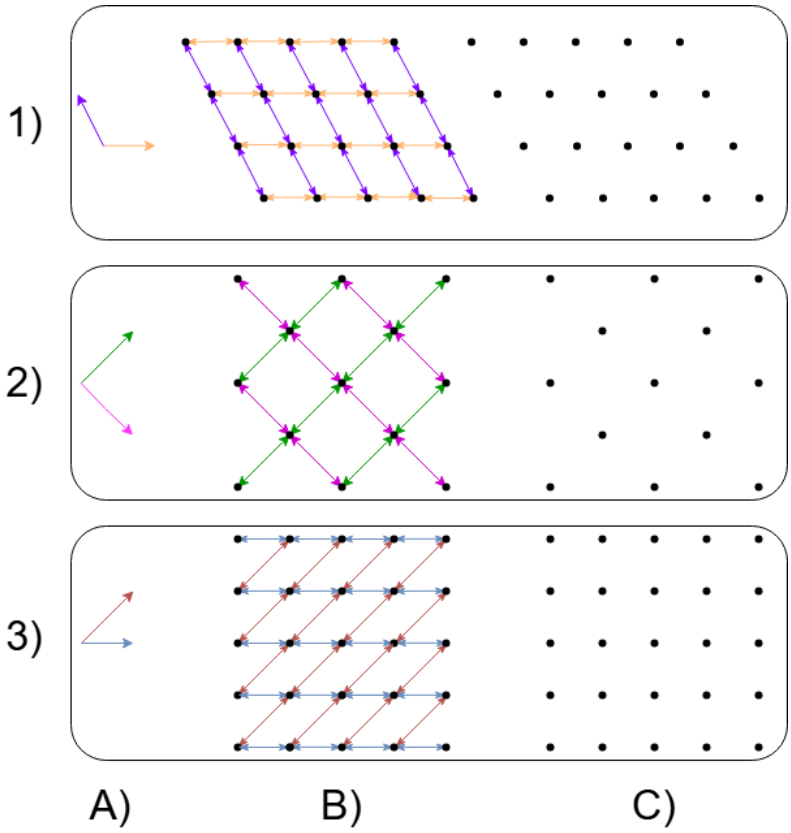

Figure 2.1: Three different two-dimensional examples of lattices.

Definition 3 (Learning with Errors (LWE) [4]) *The Learning with Errors problem is a problem that is solved by finding the secret* $\boldsymbol{s} \in \mathbb{Z}_q^n$*, where* $\boldsymbol{s}$ *is a vector with* $\boldsymbol{n}$ *elements* $(n \geq 1)$ *in the set of integers modulo* $\boldsymbol{q}$ $(q \geq 2)$*. A vector* $\boldsymbol{a} \in \mathbb{Z}_q^n$ *is generated uniformly at random and an error* $\boldsymbol{e} \in \mathbb{Z}_q$ *is generated from a* χ*-distribution. Together they create a tuple* $(a, \langle a, s \rangle + e)$ *denoted as* $\mathcal{A}_{s,\chi}$*, where* $\langle a, s \rangle = \sum_i a_i s_i$*.*

The application of LWE involves generating multiple instances of $\mathcal{A}_{s,\chi}$ over the secret key $\mathbf{s}$, resembling Gaussian elimination. However, the error term $\mathbf{e}$ introduces noise, thereby increasing the problem's computational complexity. Regarding this difficulty, in Regev's [4] paper where he first introduced a public-key cryptosystem based on the LWE-problem, states that its security is based on the worst-case quantum hardness of the lattice problems GapSVP and SIVP. SVP stands for Shortest Vectors Problem, "Gap" adds constraints to the problem and for SIVP the "I" stands for independent.

2.1.2 Ring-Learning with Errors

Ring-learning with errors (RLWE) follows the definition of LWE, but it establishes further traits to the mathematical theory, namely ring theory. As Lyubashevsky et al. [7] state, RLWE is parameterized by a number field K, a ring of integers $\mathcal{R}$ and an integer modulus $\mathbf{q}$. Additionally, Lyubashevsky et al. focus on the duality within lattices, hence why in their definition of the secret $\mathbf{s}$ they write that $\mathbf{s} \in \mathcal{R}_q^\vee$. However, Lyubashevsky et al. clarify that for the case of the m^{th} cyclotomic ring where $\mathbf{m}$ is a power of two, then $\mathcal{R}^\vee$ is just a scaling of $\mathcal{R}$. This is going to be further explained in Section 2.8.

Definition 4 (Ring-Learning with Errors (RLWE) [7]) *For a secret* $\boldsymbol{s} \in \mathcal{R}_q^\vee$*, an error distribution* Ψ *over* K*, and a sample from the ring–LWE distribution* $\mathcal{A}_{s,\Psi}$ *over* $\mathcal{R}_q \times \mathbb{T}$*, where* $\mathbb{T} = K_{\mathbb{R}}/\mathcal{R}^\vee$*,* $\boldsymbol{a}$ *is generated uniformly at random,* $\boldsymbol{e}$ *is generated from the error distribution* Ψ *and* $(a, b = (a\mathring{u}s)/q + e \mod \mathcal{R}^\vee)$ *is outputted. The problem is stated as: Given access to arbitrarily many samples of* $\mathcal{A}_{s,\Psi}$*, find* $\boldsymbol{s}$*.*

Definition 4 is a concise version of the definition given by Lyubashevsky et al., but if the reader could follow Definition 3, simply replace most of the components to that of the components of a ring.

2.1.3 CKKS

CKKS (Cheon-Kim-Kim-Song) is a leveled-homomorphic encryption scheme for arithmetic of approximate numbers, also known as HEAAN [8]. Similar to other HE schemes, CKKS incorporates errors to its scheme, more specifically it incorporates RLWE to its scheme. Other HE schemes separate the message and the error in the plaintext, whilst CKKS combines them, $\Delta \cdot m + e$, which can be seen in Figure 2.2. This does not only contribute to the security of the schemes – based on the hardness assumptions of LWE and RLWE problems – it also contributes to the approximation of numbers [8]. Moreover, other than the distinct structure of plaintexts, several components that make CKKS possible are for example: Re-

linearization, key generation, scaling, rescaling, encoding and decoding vectors to and from polynomials. We will cover scaling, encoding and decoding in this section, whilst relinearizaion and key generation will be covered in their respective Sections 2.8.4 and 2.8.2.

The most complex component of CKKS, besides RLWE, is the mathematical theory of cyclotomic rings. Let $\mathcal{R} = \mathbb{Z}[X]/\Phi_m(X)$ be a cyclotomic ring, $\mathbb{Z}[X]$ is an integer polynomial ring and $\Phi_m(X)$ is the m^{th} cyclotomic polynomial, together they create a cyclotomic ring. The encoding and decoding of CKKS uses the cyclotomic ring's properties of isomorphism and the m^{th} primitive roots of unity to transform complex vectors to polynomials and then polynomials back to complex vectors, this is called the complex canonical embedding map σ. Let $m(X) \in \mathcal{R}$ be an encoded plaintext polynomial, the first step to decoding it would be to use the canonical embedding map $\sigma : \mathcal{S} \to \mathbb{C}$, where $S = \mathbb{Z}[X]/\Phi_M(X)$ is an arbitrary element from the ring of integer polynomials [8]. Lastly, natural projection $\pi : \mathbb{H} \to \mathbb{C}^{\phi(M)/2}$ is used on $\sigma(m(X))$ to get the decoded complex vector plaintext $z \in \mathbb{C}^{\phi(M)/2}$. Furthermore, $\mathbb{H}$ is the space of Hermitian-symmetric complex vectors indexed by $\mathbb{Z}^*_{\mathbb{M}}$, formally written as $\mathbb{H} = \{(z_j)_{j \in \mathbb{Z}^*_{\mathbb{M}}} : z_{-j} = \bar{z_j}, \forall j \in \mathbb{Z}^*_{\mathbb{M}}\} \subseteq \mathbb{C}^{\phi(M)/2}$ [8]. Note that there are two different "phi" notations, ϕ which is Euler's totient function and the m^{th} cyclotomic polynomial Φ_m. Additionally, recall that the $\mathbf{m}$ for the cyclotomic polynomial is a power of two, thus turning the notation of the ring to $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$. Moreover, $\mathbf{q}$ is the modulus in the ciphertext space, recall $\mathbf{q}$ from $\mathcal{R}_q$ which takes the polynomial from the ring modulo $\mathbf{q}$.

After encoding is complete, something that is not as complex as encoding and decoding is the scaling of the messages. In Figure 2.2, there is a visual representation of how a ciphertext is structured. The scale is represented with Δ, which is from 2^{Δ}. While in the figure, the message $\mathbf{m}$ is multiplied with Δ, in actuality it is multiplied with 2^{Δ}. However, using only Δ simplifies it, and will be used throughout the paper. Nevertheless, in the figure we can see how once the message $\mathbf{m}$ has been multiplied with the scale Δ, it takes over most of the space from the budget. This is what allows CKKS to be used for *approximate* arithmetic, the message is scaled up and therefore the error has less of an impact to the actual message once it is decrypted; yet, the error still adds security to the scheme.

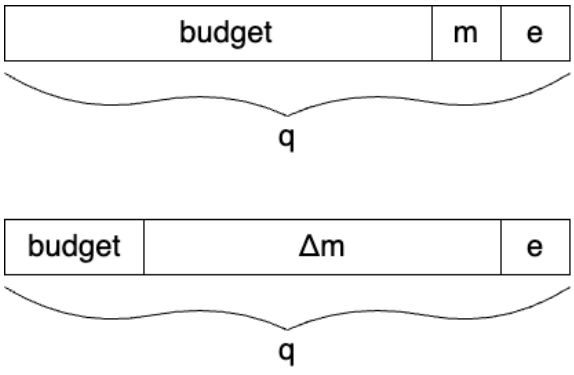

Figure 2.2: Visual representation of a ciphertext's structure

2.2 Word Embeddings

On a high level, Word Embeddings are a way to represent the semantic meaning of textual data using n-dimensional vectors. It is an emerging field that has had many leaps in the last decade within Natural Language Processing (NLP), these leaps arose from the need for a structured and mathematical representation of natural language. At first glance word embeddings sound like a way to create an embedding for individual words, which was the case in the earlier models.

Earlier models like *Word2Vec* [9] created vectors for individual words. The outputs of Word2Vec are very concise dense representations, compared to the earlier works that typically outputted sparse representations. A dense representation (dense vector) is an n-dimensional vector that is mostly filled with non-zero entries, whereas sparse representations (sparse vectors) are filled with mostly zero elements and in the case of one-hot encodings only one element is non-zero. One-hot encoding is a method in which we have a vocabulary $\mathbb{V}$, which is a list of all the words that occur at least once in the training text corpus. Then, for each instance of the word, the embedding will be a vector with all but one zeros, which means that the number one is inserted in the same index as it appears in the vocabulary $\mathbb{V}$. For example, given the vocabulary $\{cat,\ dog,\ mouse,\ squirrel,\ hedgehog\}$, the one-hot encoding of the word hedgehog will be the vector $\{0, 0, 0, 0, 1\}$.

Word2Vec was presented in a paper by Mikolov et al. [9] and was a step forward in semantic encoding of natural language. While earlier models had aimed to create semantic word embeddings from natural language, they were very computationally expensive. Word2Vec aimed to preserve estimation accuracy with a focus on making it less computationally expensive by building a simpler model. The results of Word2Vec was efficient estimation of semantic meaning of words along with algebraic consistency. Mikolov et al. give the example of taking the embedding of words (denoted by $E(X)$ where X is the word and E is the embedding function) and applying mathematical operations that retain semantic accuracy. Mikolov et al. give the example that the embedding vector resulted from $E(King) - E(Man) + E(Woman)$ was closest to the embedding vector of $E(Queen)$. This example, along with many others such as capitals in countries, illustrates the idea that the Word2Vec model achieves true semantic meaning in its embedding vectors.

The field of word embeddings has advanced from the Word2Vec model to more encompassing models, such as BERT, which was presented in a paper by Devlin et al.[10]. BERT allows arbitrary length embeddings that encode contextual meaning of words, as opposed to Word2Vec in which the same word always resulted in the same vector regardless of surrounding context.

Our focus is textual data, however our work can be implemented using any embedding model that outputs a vector of real numbers. Nowadays, creating embeddings from natural language (text and audio) and imagery is also possible due to advances in the field. Examples of some notable embedding models that deal with textual, image and audio input data are:

- Text Embedding model: BERT[10]

- Image Embedding models: DINOv2[11] & CLIP[12]
- Audio Embedding model: wav2vec 2.0[13]

2.2.1 Inversion Attacks

Word embeddings do not protect privacy and should be seen as a way of mapping from textual space to vector space. Inversion attacks are attacks in which the adversary has access to some embeddings and wishes to retrieve the original plaintext data from the embeddings. By using inversion attacks, it is possible to map back data from vector space into regular text with high accuracy [14].

In a paper by Morris et al. [15], they created and presented *Vec2Text*. Vec2Text is a model trained on text and embedding pairs. When running Vec2Text it starts with a guess string which is the initial hypothesis, thereafter the model iteratively improves the guess. It does this by creating an embedding for the initial hypothesis, then compares it against the target embedding and modifies the guess for the next iteration until it finds the match. After 50 iterations and usage of beam search, Morris et al. found that 92% of the time they were able to retrieve the original text. For Vec2Text to work as intended the adversary must have access to the same embedding model that generated the target embedding. Therefore, the adversary only needs black box access to the embedding model.

Figure 2.3 depicts the guessing process. Note that the embeddings are shown in 3 dimensions for simplicity, because embeddings are rarely encoded in 3-dimensional vectors. The guesses are generated using a model trained on embedding-text pairs as mentioned before.

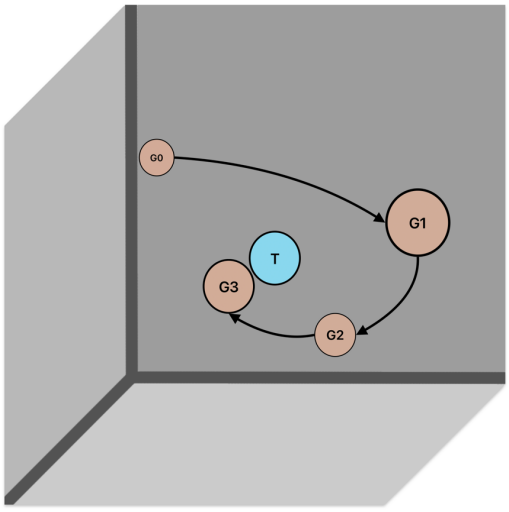

Figure 2.3: Figure inspired by the paper detailing Vec2Text [15].

A protection method that Morris et al. [15] mentioned was to add slight uniformly distributed Gaussian noise to the vector dimensions, which impaired inversion accuracy. However, Morris et al. showed that they could still achieve good similarity search performance to a certain threshold of noise. This is because noise does not

uniformly affect the semantics encoded in the embeddings, but it is theorized to affect components such as commas and other symbols that do not encode crucial semantics.

Furthermore, in a paper by Zhuang et al. [16] they analyzed the Vec2Text attack and proposed a new mitigation technique of their own. The technique Zhuang et al. proposed was to apply a transformation to embeddings after being computed. They reason that this transformation could act as a private key for the user. This is however not cryptographically secure, they did not expand with a cryptanalysis of their method.

2.3 Vector Similarity Search

Vector similarity search (nearest neighbor search) is a field within mathematics and is used in computer science for comparison between vectors to measure their similarity. This can be a similarity between the vectors' angle (cosine similarity) or a measure of distance between the vectors' ending position (euclidean distance). Vector similarity search is widely used in NLP and large-scale data applications, such as recommendation systems, where the goal is to identify items the the user might be interested in based on preferences based on their past actions. However, we are more interested in semantic search with textual data that is encoded into vectors, for which the incoming query embeddings can be compared against the stored embeddings. This will retrieve relevant data under the assumption that the vector embedding model encodes semantic meaning into its vectors, i.e. semantically similar texts are close in vector space.

The naive approach to nearest neighbor search is to compute the distance metric between the query vector and every vector in the dataset and then take the top results. This is infeasible when the dataset is large and fast lookup time is important. There is plenty of research in this area on which data structures and techniques to use for efficient computations. One example of solving this issue is to use approximate nearest neighbor(ANN) search, in order to efficiently find candidate vectors and then compute similarity metrics.

2.4 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is a type of algorithm that is designed for fast approximate similarity search. These algorithms are specifically for approximate nearest neighbor (ANN) search. ANN algorithms are similar to similarity search, because they deal with approximations instead of **absolute** nearest neighbor search. An example of a widely adopted use-case is near duplicate search, whether it is documents, images or any other media. We will be focusing more on LSH of vectorized data. There are two broad categories of LSH families which are data-independent and data-dependent algorithms which we will explain in the two next subsections.

2.4.1 Data-independent LSH

As the name suggests data-independent LSH is comprised of LSH algorithms which do not account for the data being hashed in their hashing methods. Two LSH families of note are the Hyperplane LSH [17] and E2LSH [18] (the naming of E2LSH was made after the paper was written). In Hyperplane LSH the process is to generate n random vectors of dimensionality d, we call these the bases for the LSH function and will act as hyperplanes. Then for an incoming vector $\hat{a}$ of dimensionality d that will be hashed we compute the dot product between $\hat{a}$ and the normal of each base from the list of bases. If the dot product is positive, insert a 1 and if it is negative, insert a 0 in the resulting hash list. The bases list must of course have the same entries and order for each subsequent LSH call. Hyperplanes LSH works well for cosine similarity and has been shown to outperform many data-dependent implementations given enough bases [19]. When dealing with euclidean distances there exists LSH families, with one notable early work namely E2LSH [18], which will generate n random vectors of dimensionality d we call these the bases for the LSH function and will act as projection planes. Then for incoming vector $\hat{a}$ of dimensionality d that will be hashed we compute the resulting point after we project the vector on the projection plane. This resulting point is assigned to a bucket (ID) depending on the region that it inhabits on the projection line. This is done for each projection plane and results in a hash list filled with bucket IDs.

There have been improvements in data-independent LSH algorithms such as the work presented in a paper by Andoni et al. [20] which works well for cosine similarity and Mulitprobe LSH presented by Lv et al. [21] for euclidean distance.

2.4.2 Data-dependent LSH

Data-dependent LSH is as it sounds, the opposite of data-independent LSH. The LSH function learns the dataset and creates optimal partitions or bases given the dataset. This means that it will work best for static datasets that are not manipulated after calculating LSH for them. Many algorithms have been created to solve for this issue, with many earlier iterations being very computationally expensive. An algorithm that is the data-dependent equivalent of the hyperplane LSH is Density Sensitive Hashing presented by Lin et al. [22] for cosine similarity. For euclidean distance a paper presented by Andoni and Razenshteyn details an algorithm [23].

2.5 Vector Stores

Vector stores (vector databases) are databases in which the primary objects to be stored are vectors, they often allow flexible metadata to accompany the vectors. The lookup mechanism is often a vector similarity metric, such as cosine similarity or euclidean distance to name a few. They often rely on ANN algorithms for quick lookup given a query. Vector stores are widely used to store embeddings for future lookup. It is possible to conduct semantic search using vector databases, which is just vector similarity search but often referred to as semantic search, because the embeddings encode semantics.

2.6 Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) is a technique presented in a paper by Piktus et al. [24], the aim was to efficiently increase the ability of generative language models to perform tasks that require knowledge. They did this by combining parametric and non-parametric data; parametric being the generative language models while non-parametric being a retrieval mechanism of data stored in a vector database. They present the issues that language models have which are an inability to revise or update data and hallucinations; hallucinations being that the pre-trained language models can fail to answer correctly and confidently return incorrect information. Prior works resulted in very task specific implementations which was something Piktus et al. aimed to improve upon. This led to the work done on RAG which outperformed task specific implementations while unifying the solution into a generalized model architecture for question and answering (Q&A). Thus Retrieval Augmented Generation was found to be a very effective and novel approach to achieving better Q&A functionality from generative language models while minimizing hallucinations.

The essence of RAG is to have a vector store with embeddings which is used when given a query to retrieve relevant information with semantic search. Then feeding the retrieved data along with the query to get an answer based on the context data found in the document vector store. This architecture is widely used now and can be seen in AI chat bots or even when ChatGPT is searching the web and inputting non-parametric data into the model for better Q&A functionality.

2.7 Libraries

This section will give a contextual and technically explanation of different libraries that we used.

2.7.1 MicrosoftSEAL

MicrosoftSEAL, also known as SEAL, is an open-source collection of homomorphic encryption libraries written in modern standard C++ [25]. SEAL provides three different encryption schemes to be used: BFV [26], BGV [27] and CKKS. SEAL also provides control of high level functions to separate different sequences in the code in order to perform benchmarks.

2.7.2 TenSEAL

Taken directly from the TenSEAL GitHub, “TenSEAL is a library for doing homomorphic encryption operations on tensors, built on top of Microsoft SEAL. It provides ease of use through a Python API, while preserving efficiency by implementing most of its operations using C++” [28]. It does not allow for fine grained control over some operations as MicrosoftSEAL does however it is highly optimized and provides more than sufficient runtime performance.

2.7.3 Sentence Transformers

Sentence Transformers (a.k.a SBERT) is a highly established module for access and training of embedding and reranker models[29]. However, access of embedding models is what is relevant to this thesis.

2.8 CKKS in SEAL

This section will explain how CKKS works in MicrosoftSEAL to act as the fundament for understanding the results and discussions in this paper. The primary objects to understand are the parameters for creating a context in SEAL. Context is what contains all they keys as well as the parameters poly_modulus_degree and coeff_modulus, which will be explained in Section 2.8.1. The public key is used for encryption, the secret key is used for decryption, the relinearization key is used for relinearization and the Galois key is used for rotation. How plaintexts are encrypted and how ciphertexts are decrypted will be explained in Section 2.8.2. In addition, both the relinearization key and the Galois key will be explained in their respective subsections 2.8.4 and 2.8.3. It should be noted that, from the theory about CKKS in Section 2.1.3, the notation used for the cyclotomic rings was $\mathbb{Z}[X]/\Phi_m(X)$, in SEAL it is converted to $\mathbb{Z}_q[x]/(x^N+1)$. This conversion is two-fold, first the $\mathbf{q}$ is added to the ciphertext space, $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$, then the modulus polynomial is changed from $\Phi_m(X)$ to (x^N+1), which is the same as $\Phi_{2N}(X)$.

2.8.1 Poly_modulus_degree & Coeff_modulus

The poly_modulus_degree, as the name suggests, is the modulus degree for the polynomials, and needs to be a power of two [30]. The coeff_modulus is a set of prime numbers, where the value given in the parameter decides the amount of bits the prime number has. When we multiply together all the prime numbers from the coeff_modulus we get our $\mathbf{q}$, the modulus in the ciphertext space. Because $\mathbf{q}$ is the multiplication of several primes of bit-sizes usually ranging from 20-60 (20 being the minimum and 60 being the maximum bit-size) [30], SEAL implements the "FullRNS" [31] variant of the Fan-Vercauteren scheme to handle such large values for homomorphic computations. Without going into great detail as to how the Residue Number System (RNS) works, we know it is used in order to greatly increase performance. In particular, multiplying two polynomials of degree N (poly_modulus_degree) by naive convolution is a complexity $O(N^2)$, but using Number Theoretic Transform (NTT) reduces it to a quasilinear complexity $O(N \log N)$ per prime [32]. Thus a full RNS multiplication costs $O(L \cdot N \log N)$, where L is the number of prime elements in the coeff_modulus. Again, without going into detail as to what NTT is, we know that it is based on Fast Fourier Transforms (FFT), in other words, a modular version of the FFT. NTT is applied separately to each prime element in the coeff_modulus, $q = \{q_0, q_1, ..., q_L\}$. Therefore, the more elements in the coeff_modulus, the longer the computational time, because NTT is applied to more elements.

Moreover, a visual representation of the coeff_modulus and its modulus chain is shown in Figure 2.4. The figure includes five prime elements, where the last one is

named the “special prime”. In the example code from MicrosoftSEAL, it is stated that, “The special prime should be as large as the largest of the other primes in the coeff_modulus, although this is not a strict requirement” [33]. This is further supported in a paper by Blatt et al. [34], where they argue that selecting primes close in size to the ciphertext scale—including both the first and special primes—helps minimize approximation error introduced during RNS rescaling.

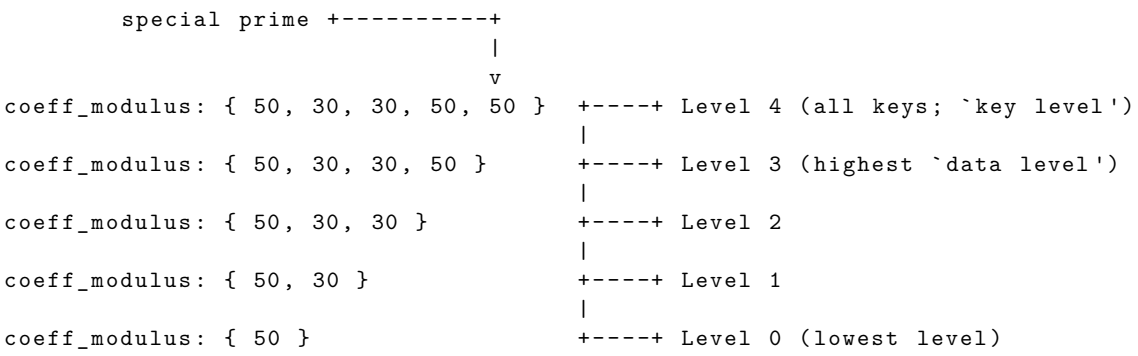

Figure 2.4: Representation of the modulus chain and its levels, taken from 3_level.cpp [33].

Additionally, there is a correlation between the coeff_modulus, the scale Δ and the amount of computations performed on the ciphertext. CKKS uses something known as rescaling, which Cheon et al. state that it preserves “the precision of the message after approximate computation” [8]. Rescaling is performed by taking the ciphertext and its scale then dividing it with a prime from the modulus chain, thus reducing the ciphertext’s scale. This is done for two reasons, first reason being that if the ciphertext’s scale is larger than 2^{60}, then the last prime is not enough and the ciphertext cannot be decrypted. Second reason being that the dropped primes from the modulus chain, are also not accounted for in the RNS. Therefore, less computations and faster performance for subsequent sequences and homomorphic operations. A toy example of how rescaling works is shown in Figure 2.5. The figure contains pseudocode where two ciphertexts are encoded and encrypted with a scale of 2^{60}. The two ciphertexts are then multiplied and rescaling is performed twice on the product ciphertext. Lastly, decryption and decoding is performed. The figure also contains the corresponding structure of the modulus chain as rescaling is performed.

```
(1)     encoder.encode(data_0, scale = 2^60, plaintext_0);
(2)     encryptor.encrypt(plaintext, ciphertext_0);

(3)     encoder.encode(data_1, scale = 2^60, plaintext_1);
(4)     encryptor.encrypt(plaintext_1, ciphertext_1);

(5)     evaluator.multiply(ciphertext_0, ciphertext_1, ciphertext_product);

(6)     evaluator.rescale_to_next_inplace(ciphertext_product);
(7)     evaluator.rescale_to_next_inplace(ciphertext_product);

(8)     decryptor.decrypt(ciphertext_product, plaintext_product);
(9)     encoder.decode(plaintext_product, data_product);

+-------------------------------------------------------------------------------+

        Modulus Chain structure:
                                  v
(6)     coeff_modulus: { 60, 31, 31, 60 } // Prime is dropped and used for rescaling
                            v
(7)     coeff_modulus: { 60, 31, 60 } // Prime is dropped and used for rescaling
                          v
(8-9)   coeff_modulus: { 60, 60 } // Prime is used for decryption and decoding.
```

Figure 2.5: Toy example of performing two rescaling operations, accompanied by a structural example of the modulus chain.

2.8.2 Encryption & Decryption

From the papers we have researched that cover CKKS encryption, we have not found one that exactly describes the schemes structure and components. Instead, the original CKKS paper by Cheon et al. [8] explains a generalized construction based on the BGV scheme [27]. Pair that with the limited descriptive documentation of how MicrosoftSEAL implements CKKS, we created Figure 2.6 which is how we have interpreted the different components, encryption and decryption.

$$secretkey = s \leftarrow R_q, \{-1, 0, 1\} \tag{1}$$

Example of a secret key polynomial with a degree of four and mod 13:

$$s(X) = -1 * X^0 + 0 * X^1 + 1 * X^2 + 1 * X^3$$
$$s(X) = (12, 0, 1, 1)$$

$$publickey = \begin{cases} p_0 = -a \cdot s + e \\ p_1 = a \end{cases} \tag{2}$$

$$ciphertext = \begin{cases} c_0 = -a \cdot s + \Delta \cdot m + e_0 \\ c_1 = a + e_1 \end{cases} \tag{3}$$

$$c_0 + c_1 \cdot s = -a \cdot s + \Delta \cdot m + e_0 + (a \cdot s + e_1 \cdot s) = \Delta \cdot m + (e_0 + e_1 * s) = \Delta \cdot m + e \tag{4}$$

Figure 2.6: Mathematical computations to encrypt and decrypt, as well as an example of a secret key.

First, the secret key is a polynomial generated from the ring $\mathcal{R}_q$ with coefficients of {-1,0,1}. The coefficients are from a function, *sample_poly_ternary* from keygenerator.cpp [33], that generates a uniform ternary polynomial and stores it in RNS representation. As for the use of a ternary polynomial, the reasoning is stated by Albrecht et al. [35] in their technical report about standardization for HE. They very specifically state, "...choosing an even smaller secret key has a significant performance advantage. For example, one may choose the secret key from the ternary distribution (i.e., each coefficient is chosen uniformly from {1,0,1})". This is how MicrosoftSEAL has implemented their secret key generation, for both performance and security reasons.

Additionally, in Figure 2.6 we show an example with a degree of four and a modulus of 13, in practice these values would be much larger, especially the modulus. Second, the public key is two components. Both components contain the term **a** which is generated uniformly at random from the ring–LWE distribution $\mathcal{A}_{s,\Psi}$. Here we used Ψ correctly because Cheon et al. refer and correlate it to a Gaussian distribution. This is because the Gaussian distribution is also used to generate the error **e**, which is also in Figure 2.6 and a part of the public key's first component. The way a message is encrypted after being scaled is by simply adding it to the first component of the public key. The decryption calculations are then shown in the final equation at the bottom of Figure 2.6.

2.8.3 Rotation

Rotation is an operation in MicrosoftSEAL that shifts a ciphertext vector n-amount of steps to the left or to the right, an example is shown in Figure 2.7. In order to perform rotation, something known as the Galois key needs to be generated and used on the ciphertext. The properties of the Galois key correlate with how plaintext vectors are encoded and then encrypted into ciphertext polynomials, recall the isomorphic properties of cyclotomic rings. Isomorphism was not fully explained and covered in Section 2.1.3, same thing will be true for Galois automorphism, which is what allows ciphertext polynomials to be **rotated**. The reason being, for both cases, that it involves a lot of advanced group theory, which would be a lot to cover and not necessary for the reader to understand how MicrosoftSEAL and CKKS work.

```
evaluator.rotate_vector(ciphertext, step, galois_keys, rotated_ciphertext);
```

$$ciphertext = \{c_0, c_1, ..., c_k\} \quad \rightarrow \quad rotated\_ciphertext = \{c_1, c_2, ..., c_k, c_0\}$$

Figure 2.7: Example of a ciphertext being rotated one step to the left, with example code from MicrosoftSEAL.

Rotation is applying a function κ_k to an encoded and encrypted plaintext vector $m(X)$. The rotation is applied to a ciphertext, however it is easier to understand and visualize it in the plaintext world. Therefore, the example of rotation in Figure 2.8 has a vector plaintext $m_i(X)$ which contains four elements. Each vector

element, once encoded and encrypted, is mapped to the roots of unity ζ^{2i+1}, where i corresponds to the index of the vector element in the vector. Once the Galois key κ_k is applied on to the ciphertext, each root of unity ζ^{2i+1} is mapped to $\zeta^{k(2i+1)}$, effectively shifting the slots. For example, if $k = 3$ then m_0 would be mapped to m_1, because m_1 started with ζ^3, which is what m_0 will have after rotation, $\zeta^{3(2\cdot 0+1)} = \zeta^3$. In a sense, each element becomes a temporary copy of another elements value.

$$
\begin{aligned}
m_i(X) &= [m_0, m_1, m_2, m_3] \\
&\equiv [m_0 \mod (x-\zeta^1),\ m_1 \mod (x-\zeta^3), \\
&\quad\ m_2 \mod (x-\zeta^5),\ m_3 \mod (x-\zeta^7)]
\end{aligned} \tag{1}
$$

$$
\kappa_k(m(X)) \to m(X^k) \mod \Phi_m(X) \tag{2}
$$

$$
\begin{aligned}
\kappa_k(m_i(X)) \equiv [m_0 &\mod (x-\zeta^{1k}),\ m_1 \mod (x-\zeta^{3k}), \\
m_2 &\mod (x-\zeta^{5k}),\ m_3 \mod (x-\zeta^{7k})
\end{aligned} \tag{3}
$$

Figure 2.8: Example of vector elements mapped to roots of unity ζ^{2i+1} where $N = 4$, from $X^N + 1$.

Given that k is the deciding factor of how and where elements are mapped, the question remains how it is translated to n-amount of steps to either left or the right. In MicrosoftSEAL, specifically `galois.cpp` [33], the value for k is decided by the formula $k = 3^s \mod 2N$, where s is the amount of steps. If the s is positive then it corresponds to a left rotation, if it is negative then it corresponds to a right rotation. The value 3 in the formula corresponds to the generator of the multiplicative group $\mathbb{Z}^*_{2N}$.

2.8.4 Relinearization

Multiplying two ciphertexts of size two, c = $\{c_0, c_1\}$ and d = $\{d_0, d_1\}$, produces a ciphertext of size three, $P = (p_0, p_1, p_2) = \{c_0d_0, c_0d_1 + c_1d_0, c_1d_1\}$. This larger ciphertext of size three must be relinearized back to a size of two with a relinearization key, $relinearize(P) = c' = (c'_0, c'_1)$, otherwise it cannot be decrypted. MicrosoftSEAL's functions, for example the function *rotate_internal()* which calls the function *apply_galois_inplace()* in `evaluator.cpp` [33], explicitly check if the size of the ciphertext, which rotation is going to be performed on, is greater than two. Chen et al. [36] define the relinearization key as $relin\_key = ([-(as + e) + s^2]_q, a)$, where $a \leftarrow \mathcal{R}_q$ and $e \leftarrow \mathbb{Z}_q$ for a χ-distribution. Chen et al. also demonstrate how the relinearization key is used and how the decryption is performed, both of which are shown in Figure 2.9.

$$c'_0 = c_0 + c_2 evk[0]$$
$$c'_1 = c_1 + c_2 evk[1]$$
Decryption of c' would yield:
$$c'_0 + c'_1 s = c_0 + ((as + e) + s^2)c_2 + c_1 s + ac_2 s = c_0 + c_1 s + c_2 s^2 - ec_2.$$

Figure 2.9: The use of a relinearization key and the equation of decrypting the relinearized ciphertext [36].

However, as can be seen from Figure 2.9, the resulting decrypted message is left with a large negative error that is multiplied with the element c_2. Recall that c_2 is the same as p_2 from the first example, which from Figure 2.6 means it is made out of the second element from both ciphertexts multiplied together. Therefore, not only does c_2 multiply with an error, but c_2 in and of itself contains two errors multiplied with each other, resulting in a lot of noise in the final decrypted message. The solution that Chen et al. provide is a decomposition of c_2 with base w, shown in Figure 2.10, which means that the relinearization key is also decomposed.

$$relin\_key[i] = ([-(a^{(i)}s + e^{(i)}) + w^{(i)}s^2]_q, a^{(i)})$$
$$c'_0 = c_0 + \sum_{i=0}^{\ell} evk[i][0]c_2^{(i)}$$
$$c'_1 = c_1 + \sum_{i=0}^{\ell} evk[i][1]c_2^{(i)}$$
Decryption of c' would yield:
$$c'_0 + c'_1 s = c_0 + c_1 s + \sum_{i=0}^{\ell} c_2^{(i)} w^{(i)} s^2 - \sum_{i=0}^{\ell} c_2^{(i)} e^{(i)}$$

Figure 2.10: Decomposition of the relinearization key and the resulting error [36].

Essentially, relinearization is required and used for two reasons. The first reason, stated by Chen et al. [33], is that that the scheme loses its compactness property. In other words, the circuit or the evaluating function should not depend on the ciphertext. Therefore, returning to a size of two, before performing another homomorphic operation or computation, ensures the compactness property of CKKS in MicrosoftSEAL. The second reason is that the noise would grow exponentially with every homomorphic multiplication of non-relinearized ciphertexts.

3

Process and Method

This chapter will detail our workflow and follows a somewhat chronological order. We start by presenting our initial experiments for exploration of the viability within the use-case. Then we move on to presenting the work we did on creating a database that provided storage and efficient retrieval of encrypted embeddings, where Python was used along with the HE library TenSEAL. Additionally we present a formalized version of the client & server architecture. Lastly we detail the work we did using C++ and MicrosoftSEAL to explore how much better the performance of homomorphic operations could be within our use-case.

3.1 Preliminary experiments for cosine similarity using HE

Before theorizing about the applicability of HE in real applications we had to conduct simple benchmarks to grasp the temporal context which we will be working in. The benchmark script is straightforward and was run on Apple Silicon chips (the M1 Pro and M3 chips) with hardware acceleration enabled on the PyTorch platform for encoding of the word embeddings. When prototyping we decided on using the SentenceTransformers (a.k.a SBERT) library and we chose to follow the Beir[37] GitHub repository to find datasets. Furthermore, in our prototype we chose to use the SCIDOCS dataset from the Beir GitHub which consisted of a corpus and a set of queries both separated into their respective JSONL (JSON lines) files. We then proceeded to extract the text data from these two data files and created embeddings for each corpus entry (which consisted of a title and a roughly paragraph sized body) and query.

TenSEAL was used for homomorphic encryption, and CKKS was the HE scheme that was used when computing the benchmarks for the distance metrics described in the next two paragraphs. In the homomorphic encryption domain we are limited to addition and multiplication operations. Division and square roots in the ciphertext domain are not possible in HE schemes. However there are methods for estimation of division and the square root which are costly and not exact in their accuracy.

When we computed the cosine similarity for two vectors, we prepared them by normalizing the embeddings before encryption. This holds because the cosine similarity is reduced to the dot product between vectors when normalized [38]. By normaliz-

ing the embeddings we avoid division between ciphertexts and that way the cosine similarity can easily be computed using the dot product between the two encrypted vectors. We chose to avoid division, because the more preparations we do before encryption, leads to less computation operations in the homomorphic space. After we managed to correctly compute cosine similarities, we benchmarked the code for performance. The data we measured compared time needed to compute each cosine similarity score for ciphertext pairs, the resulting cosine similarity score as well as the plaintext cosine similarity score. The measured data was then averaged to retrieve the average computation time for the cosine similarity scores along with the average percent difference between ciphertext cosine similarity and plaintext cosine similarity.

When computing the euclidean distance, we settled on the squared euclidean distance because it was simpler to implement using HE compared to the non-squared euclidean distance [38]. We reason that after decryption the root can easily be calculated if needed, if not the square will still have the same order as the root when sorting between decrypted distances. No preparations were needed on the plaintext vectors for the squared euclidean distance. For vectors $\hat{a}$ and $\hat{b}$:

$$c = \hat{a} - \hat{b}$$
$$Squared\ Euclidean = c \cdot c$$

After experimenting with the squared euclidean we ran some benchmarks. The measured data compared the time needed to compute each squared euclidean distance for ciphertext pairs, the resulting ciphertext squared euclidean distance as well as the plaintext squared euclidean distance. This data was then averaged to retrieve the average computation time for the distance along with the average difference between ciphertext squared euclidean distance and plaintext squared euclidean distance.

3.2 Client and Server: Prototype

After ensuring that our similarity computations were in fact correct, and verifying that the use-case of computing vector similarity in the homomorphic encryption domain was viable, the next step we took was to start working on a client and server.

We tried to get funding from Chalmers to access a cloud service for the server hosting. However there was no such program available for that. There is currently no separation between the client and server, which was not the envisioned implementation nor our intention and initial goal. Therefore, we had to compromise and worked on a more simplified version of a client and server application hosted locally on the same machine. The code requires some refactoring to be separated, however this is something we have not implemented.

The creation of embeddings was the same as the previous section, SBERT was used to create embeddings from our datasets. The embedding models we used was msmarco-bert-base-dot-v5(768-dimensional embeddings). We later moved over to the model all-MiniLM-L6-v2 (384-dimensional embeddings) because it is faster and

generally outperforms the previous model. The dataset used was the same as in the previous section, namely SCIDOCS. TenSEAL was used for our homomorphic encryptions.

Efficient retrieval of relevant stored encrypted embeddings given a query was the issue we encountered before starting our work on the server. This is because it is one of the goals of vector stores. Recall that we would like to retrieve the most similar embeddings (cosine similarity) or closest embeddings (squared euclidean) upon receiving a query embedding. Imagine that a user has uploaded 5000 individual entries in the database. Without an effective way of getting the most relevant entries we would end up computing the similarity metric 5000 times per query, this is not really feasible since we don't want to scale latency and response size along with the amount of database entries.

We ended up finding one method that would allow for this, the method is Locality Sensitive Hashing (LSH) which was described in Section 2.4. Locality Sensitive Hashing pertains to a fuzzy hashing technique in which similar items are stored close together or can be turned into fingerprints in which similar vectors receive a similar fingerprint; we realized that we had two approaches we could take. One approach would be to do LSH on the plaintext, while the other would be to do it on the ciphertext. The ciphertext seemed infeasible after a little bit of modeling hence why we chose to do them on the plaintext.

We use the hyperplanes LSH and generated 384 random vectors which we use as hyperplanes for a given collection of embeddings. Note that these hyperplanes need to be persistent for correctness. Then for each embedding we calculate if the dot product between the embedding and each base's normal is negative or positive, this indicates whether the embedding is in front of or behind the hyperplane. This is used to create a 384-dimensional array of ones and zeros, which we refer to as the embedding fingerprint. The embedding fingerprint can be used to quickly search through the database for closest neighbors. The way to compare fingerprints is by checking equality per index. Then we will check which rows have the most matches with the query row. Our LSH implementation is limited to cosine queries since the fingerprint does not take into account vector length. To further extend this we would use E2LSH (presented in Section 2.4) when entries and queries are set to the squared euclidean distance. However, we did not manage to implement this in time. Furthermore, the idea is to use LSH to retrieve closest neighbors to then calculate the cosine and euclidean distance for each embedding because we want to get a sorting that is as accurate as possible. See Appendix A for our high-level suggested prototype architecture.

The database we chose was PostgreSQL due to it being open source, reliable and very extensible. We hosted our database locally in a docker container. We created the database schema in Python using SQLAlchemy and used Alembic to migrate the schema to the database, see Appendix A for schema image. Some columns can be omitted since we didn't actually use them, however their presence indicates how the app is supposed to be structured. For example, the *password*, *salt* and *runs* columns were never used even though they are a crucial component of a functioning

application. Ultimately, we focused solely on semantic search functionality.

Homomorphic Encryption allows secure similarity search over embeddings. However, storing full plaintext documents homomorphically would be inefficient and unnecessary. Therefore, embeddings are protected via HE, and full documents are protected using fast symmetric encryption using AES for practical performance. This is because we must allow for the result to be used in some way after the search has been conducted.

The next section will provide a better view of how the server operates. As stated before, there is no clear separation between server and client in the actual code, some re-factorization is needed. We find Section 3.3 to be a more clear and concise representation of the implementation's functionality.

3.3 Client and Server: Formalized

The proposed procedures are high level descriptions of the intended software implementations. They will serve as a simplified blueprint for our created work for readability. The procedures omit many application level mechanisms, such as error handling and status codes between client and server as well as concrete descriptions of storage measures that the client will implement.

The first procedure describes the initialization phase between the client and the server. The data that is required from the client is:

- AES key-generator, this is because the plaintext data should be accessible to the client after semantic search retrieval. The data stored will be plaintext natural language data, i.e. what the data is before it is turned into embeddings.
- Parameters for HE scheme initialization.
- Key-generator for the HE scheme given the set of input parameters.
- HE scheme's encryption and decryption functions.

The client will generate a set of keys using the HE scheme's key generator. These keys will be saved on the client for future usage. The next step is to generate a symmetric key for AES encryption and decryption, this is done using the AES key-generator and saved on the client for future usage.

Procedure 1 Init phase

Client Initialize

AesKeyGen // Keygen for our AES use
$KeyGen(p)$ // HE Scheme's keygen given parameters p
Enc_{pk} // Homomorphic Encryption
Dec_{sk} // Homomorphic Decryption
params // Encryption parameters

$pk, sk, ek \leftarrow KeyGen(params)$
$pk, sk, ek \rightarrow Storage$
$sk_{AES} \leftarrow AesKeyGen$
$sk_{AES} \rightarrow Storage$

The next stage for the client is to upload the document corpus that they intend to search through using HE augmented similarity search. This stage requires the client to have performed the initialization phase, the prerequisites for the client are:

- The document corpus that the user intends to upload.
- An embedding model that will be used to turn documents from natural language into vector representations.
- AES encryption function to securely store the natural language representation on the database.
- Bases (hyperplanes) to use for the LSH function.
- A function for computing the LSH fingerprint for a vector given LSH bases. The bases are used to compute the dot product between the input vector and the normal of each base used as hyperplanes, resulting in a 1 if dot product is positive (vector has the same direction as the norm, i.e. in front of the hyperplane) and a 0 if the dot product is negative (vector is behind the hyperplane).
- Normalization function for vectors, used if similarity measure is cosine. This makes computing the cosine similarity score easier when using HE operations.
- A similarity metric that the corpus will be computed on given an input query. It can be cosine similarity or the squared euclidean distance.

The user will retrieve their AES key and public HE scheme's key from storage. If the similarity metric is cosine, the user will normalize the embeddings, else the user will simply create the embeddings, these are denoted by D_{emb}. We then encrypt each embedding with our public key and our HE scheme's encryption function, the result is denoted by D_{emb*}. The last step is to generate the LSH fingerprint for the plaintext embeddings which are denoted by D_{lsh}. These are then sent to the database which will parse the data and combine the entries from each list into logical objects which are then uploaded to the database.

Procedure 2 Upload phase

Client

D // Documents user wants to upload
E // Text embedding model
AES // AES encryption function
H // LSH function
$bases$ // Hyperplanes for LSH generation
$norm$ // Vector normalization func

simMet // Similarity search metric

$sk_{AES} \leftarrow Storage$
$pk \leftarrow Storage$
if $simMet$ **is** $cosine$ **then**
  $D_{emb} \leftarrow norm(E(D))$
else
  $D_{emb} \leftarrow E(D)$
$D_{emb*} \leftarrow Enc_{pk}(D_{emb})$
$D_{lsh} \leftarrow H(D_{emb}, bases)$
$D_{aes} \leftarrow AES(D)$

$D_{emb*}, D_{lsh}, D_{aes}, clientId, simMet \rightarrow Server$

Server

$D_{emb*}, D_{lsh}, D_{aes}, clientId, simMet \leftarrow Client$

$data \leftarrow \{(d_{emb*}, d_{lsh}, d_{aes}) \mid d_{emb*}, d_{lsh}, d_{aes} \leftarrow D_{emb*}, D_{lsh}, D_{aes}\}$

$(data, simMet, clientId) \rightarrow Database$

The final stage for the client is to query the server on the uploaded document corpus. This stage requires the client to have performed the upload phase, the prerequisites for the client are:

- The query that the user intends to send.
- The embedding model that will be used to turn documents from natural language into vector representations.
- Bases (hyperplanes) to use for the LSH function.
- A function for computing the LSH fingerprint for a vector given LSH bases.
- Normalization function for vectors, used if similarity measure is cosine. This makes computing the cosine similarity score easier when using HE operations.
- A similarity metric that the corpus will be computed on given an input query. It can be cosine similarity or the squared euclidean distance.

The user will retrieve the public HE scheme's key from storage. If the similarity metric is cosine, the user will create an embedding for the query and normalize the embeddings, else the user will simply create the embeddings, these are denoted by Q_{emb}. We then encrypt the embedding (Q_{emb}) with the user's public key and the HE scheme's encryption function, the result is denoted by Q_{emb*}. The last step is to generate the LSH fingerprint for the plaintext embeddings which are denoted by Q_{lsh}. The client will send Q_{emb*}, Q_{lsh}, clientId, similarity metric and the public key.

The prerequisite for the server is to have a dot product function that will take in two ciphertexts as well as a public key and output the dot product between the two ciphertexts. The server will first parse the request and extract the Q_{lsh} field, this will then be used to query the database for n-amount of rows with the most matching elements which will be returned sorted based on the LSH matches. Then for each response row the server will compute the similarity measure, refer to the algorithm for the specifics of cosine similarity or euclidean distance. Upon having calculated the similarity score for each row the similarity measure will be added to each entry in the list of rows. This is then sent back to the client.

The client will retrieve the data from the server as well as retrieving the HE scheme's secret key from storage. Then it will decrypt the similarity measure for each element in the results list and swap the encrypted value with the decrypted value, because the encrypted value will not be needed. The client can then sort through the similarity measures and choose how to move forward. The next procedure is an example use-case for our suggested next stage.

Procedure 3 Query phase

Client

Q // Query string user wants to use
E // Text embedding model
H // LSH function
$bases$ // Hyperplanes for LSH generation, generated by user but need to be persistent across uploads/queries
$norm$ // Vector normalization func
simMet // Similarity search metric must be same for uploaded documents for correct search

$pk, ek \leftarrow Storage$
if $simMet$ **is** $cosine$ **then**
  $Q_{emb} \leftarrow norm(E(Q))$
else
  $Q_{emb} \leftarrow E(Q)$
$Q_{emb*} \leftarrow Enc_{pk}(Q_{emb})$
$Q_{lsh} \leftarrow H(Q_{emb}, bases)$

$Q_{emb*}, Q_{lsh}, clientId, simMet, ek \rightarrow Server$

Server

$\text{dot}_{ek}(u, v)$ // Homomorphic dot product of vectors u,v given an evaluation key

$Q_{emb*}, Q_{lsh}, clientId, simMet, ek \leftarrow Client$

$rows \leftarrow$ GetFromDb(Q_{lsh},simMet,lim)
$results$
for each r **in** rows
  $d_{emb*}, d_{lsh}, d_{aes} \leftarrow$ r
  if $simMet$ **is** $cosine$ **then**
    $d_{sim*} \leftarrow dot_{ek}(d_{emb*}, Q_{emb*})$
  else
    $v_{res*} \leftarrow d_{emb*} - Q_{emb*}$
    $d_{sim*} \leftarrow dot_{ek}(v_{res*}, v_{res*})$
  $results \leftarrow (d_{emb*}, d_{lsh}, d_{aes}, d_{sim*})$
$results \rightarrow Client$

Client

$results \leftarrow Server$
$sk \leftarrow Storage$
for each r **in** results
  $d_{sim*} \leftarrow$ r
  $d_{sim} \leftarrow Dec_{sk}(d_{sim*})$
  $swap(r, d_{sim*}, d_{sim})$
$sort(results, d_{sim})$

This last procedure describes a concrete use-case in which similarity search can be used to aid with information retrieval and usage by a human in a semantic search task. The client will first retrieve its secret AES key from storage. Then it will decrypt each document in the results list from Procedure 3. It will then send the documents in plaintext form along with the query to the LLM. Afterwards the LLM will return with a plaintext answer for the question given the input data.

Procedure 4 Query phase

Client - Continuation example RAG

//Assume user would like an automated answer given their query
$myLLM$ //Large Language Model hosted locally (for privacy)

$sk_{AES} \leftarrow Storage$
$context \leftarrow DEC_{AES,sk}(\{d_{aes} \mid d_{aes} \leftarrow results\})$
$(context, Q) \rightarrow myLLM$
$resp \leftarrow myLLM$

3.4 MicrosoftSEAL implementation

This section showcases the structure and sequences of our code written in MicrosoftSEAL, demonstrating the idea of our use-case and how the benchmarks were conducted. In the following page is Procedure 5, which was used to calculate the cosine similarity by performing homomorphic multiplication on encrypted normalized word embeddings, decrypting them and then later adding all the vector-elements together. The procedure has two clients and one server to contextualize each task, however, this was all done on the same machine during testing.

Firstly, the procedure needs resources and components initialized and setup. These resources and components are: The documents in a server, the query from a client, a text embedding model to convert text to word embeddings, a function that takes a word embedding and normalizes it, a function that performs the homomorphic multiplication on the ciphertexts, and finally a function that adds all the elements of a vector together. Notice how in Procedure 5, the summation function works on plaintexts and not ciphertexts, Procedure 6 will have a summation function that works on ciphertexts.

Secondly, the procedure's sequence of operations. Starting with one primary client, it generates a public key and a secret key, which is the only client able to decrypt the ciphertexts. This primary client, client A, can now share the public key to other clients who want to upload documents to the server. However, each of these clients has to first embed the documents, then normalize the embedding, encrypt it using the public key and then they can send it to the server to be stored. Now, client A can query the server, this query goes through the same procedure as the documents. Once the server receives the encrypted query it homomorphically multiplies it with every document, relinearizes it and returns every encrypted product to the client. Client A decrypts all the encrypted products, adds together the elements of each product, and then the ten highest cosine similarities are printed. In the print output, the top ten highest cosine similarities from the ciphertexts are compared to the respective plaintext cosine similarities which were generated one step prior, at step 15 of Procedure 5.

This procedure serves two purposes, the first one is to test the accuracy between plaintext cosine similarity and ciphertext cosine similarity. The second purpose is to benchmark the performance of each sequence of operations. In other words, performance benchmarks were conducted in order to deduce the viability and use of homomorphic encryption schemes for cosine similarity calculations of word embeddings.

Following Procedure 5 is Procedure 6, which uses **homomorphic** addition using rotation on the product of encrypted normalized word embeddings, in other words it performs the dot product on the ciphertexts. Rotation is used in order to perform addition of all the elements in a vector. By using the binary tree summation algorithm, the total number of rotation-operations is 10, $2^{10} = 1024$). We sum up all the 768 elements from the word embeddings together with the additional 256 zero-padding elements. Note that we used the msmarco-bert-base-dot-v5 for

these procedures, which outputs 768-dimensional embeddings. Procedure 6 also uses rescaling, in order to increase performance of rotations and relinearization.

Procedure 5 Cosine similarity with CKKS (not rotated)

Initialize

D // Documents in server
Q // Query from client
E // text embedding models
N // Normalization of vectors
mul_* // Homomorphic multiplication
relin_* // Relinearization
sum // Summation of decrypted vector elements

Client A

1: Generate a public key pk and a secret key sk
2: $pk \rightarrow Client\ B$

Client B

3: $D_{emb} \leftarrow E(D)$
4: $D_{norm} \leftarrow N(D_{emb})$
5: $D_{norm*} \leftarrow \text{Enc}_{pk}(D_{norm})$
6: $D_{norm*}, pk \rightarrow Server$

Client A

7: $Q_{emb} \leftarrow E(Q)$
8: $Q_{norm} \leftarrow N(Q_{emb})$
9: $Q_{norm*} \leftarrow \text{Enc}_{pk}(Q_{norm})$
10: $Q_{norm*}, pk \rightarrow Server$

Server

11: $c_{mul*} \leftarrow \text{mul}_*(Q_{norm*}, D_{norm})$ with pk
12: **return** $c'_{mul_*} \leftarrow relin_*(c_{mul*})$

Client A

13: $p_{mul} \leftarrow \text{Dec}_{sk}(c'_{mul*})$ with sk
14: $sim \leftarrow sum(p_{mul})$
15: $plaintext\_sim \leftarrow sum(mul(Q_{norm}, D_{norm}))$
16: $print \leftarrow topten(sim, plaintext\_sim)$

Procedure 6 Cosine similarity with CKKS (rotated)

Initialize

D // Documents in server
Q // Query from client
E // text embedding models
N // Normalization of vectors
mul_* // Homomorphic multiplication
relin_* // Relinearization
resca_* // Rescaling
sum* **// Summation of ciphertext's elements**

Client A

1: Generate a public key pk and a secret key sk
2: $pk \rightarrow Client\ B$

Client B

3: $D_{emb} \leftarrow E(D)$
4: $D_{norm} \leftarrow N(D_{emb})$
5: $D_{norm*} \leftarrow \text{Enc}_{pk}(D_{norm})$
6: $D_{norm*}, pk \rightarrow Server$

Client A

7: $Q_{emb} \leftarrow E(Q)$
8: $Q_{norm} \leftarrow N(Q_{emb})$
9: $Q_{norm*} \leftarrow \text{Enc}_{pk}(Q_{norm})$
10: $Q_{norm*}, pk \rightarrow Server$

Server

11: $c_{mul*} \leftarrow \text{mul}_*(Q_{norm*}, D_{norm})$ with pk
12: $c'_{mul_*} \leftarrow resca_*(c_{mul_*})$
13: $c''_{mul_*} \leftarrow relin_*(c'_{mul*})$
14: return $c_{sum*} \leftarrow \textbf{sum}_*(c''_{mul_*})$ **with** pk

Client A

15: $sim \leftarrow \textbf{Dec}_{sk}(c_{sum*})$ **with** sk
16: $plaintext\_sim \leftarrow sum(mul(Q_{norm}, D_{norm}))$
17: $print \leftarrow topten(sim, plaintext\_sim)$

3.5 Benchmark Methodology

This section will explain the way the upcoming benchmarks were conducted and why they were conducted in that way. Starting with the first benchmarks of Procedure 5 and Procedure 6. At the beginning, during testing and implementation, we compared and analyzed the performance and cosine similarity accuracy of different parameters. We discovered that the way we selected the scale for different parameters was not ideal and soon realized that higher scales could be implemented to

several parameters. This meant that the high cosine similarity accuracy could be achieved at a wider range than just a few specific parameters. This in turn, shifted the focus from the performance of cosine similarity accuracy and instead to the security of using high scales for high cosine similarity accuracy. We also realized that rescaling improves the performance of rotation and relinearization. Additionally, the amount of rescales, and the scale Δ, depends on the size and the amount of intermediate primes in the coeff_modulus.

The rotation benchmark was conducted in order to evaluate the viability of offloading everything to the server. It was also conducted in order to verify that a dot product could be executed in CKKS. In the end, we were able to perform a dot product with CKKS in MicrosoftSEAL and the results were sufficient and expected when compared to the performance of the Python implementation. Additionally, during the experiments of rotation, rescaling was introduced to increase performance as stated earlier. We deemed that implementing and adjusting every different parameter to this rescale property was complicated and unnecessary, one example was enough to provide support for the claim that rescaling benefits performance.

Moreover, for the plaintext documents benchmark, a paper by Kim et al. [39], gave the idea to benchmark multiplication between a ciphertext query and a plaintext document. In the implementation itself, only the query was not encrypted, but it would work the same if the documents were not encrypted as well, as long as the query then was encrypted.

4

Results

In this section we will present our results from our experiments, the TenSEAL Python prototype, the MicrosoftSEAL C++ implementation and the database using LSH.

4.1 TenSEAL Benchmarks

This section will present the benchmarks we received with our first prototype when we were exploring the use-case. Recall that TenSEAL is the Python library that allows us to use CKKS, which is provided by bindings from MicrosoftSEAL. These were taken with an Apple-Silicon chip namely the M3. These benchmarks, shown in Table 4.1, are averaged from multiple runs of 400 query embeddings compared with 400 corpus embeddings. These benchmarks are as followed:

1. Average difference in percent. This is a comparison between the plaintext result with the decrypted ciphertext result.
2. Average time for computing the similarity measure between two ciphertexts.
3. Average time to encrypt plaintext vectors, this operation includes encoding and encryption of plaintexts.
4. Average time to decrypt CKKS vectors, this operation includes decryption and decoding of CKKS vectors.

We implemented both cosine similarity as well as the squared euclidean distance as similarity measures between two vectors. Below are the benchmarks for poly_modulus_degree 8192 and coeff_modulus {60+40+40+60}.

Similarity Metric	**Cosine Similarity**	**Squared Euclidean**
Difference(%)	4.4527×10^{-4}	1.0233×10^{-5}
Computation time (seconds)	0.01552 s	0.01568 s
Encryption time(seconds)	0.0023 s	
Decryption time(seconds)	0.0007 s	

Table 4.1: Comparison of computation times, average percent difference, average encryption time and average decryption time for the cosine similarity and squared euclidean

4.2 MicrosoftSEAL Benchmarks

In this section, the benchmark results from the MicrosoftSEAL implementation will be presented. These were taken with an Apple-Silicon chip namely the M3. In Tables 4.2 – 4.6, the parameters-row contains information about the encryption scheme used. It also contains the poly_modulus_degree, followed by the coeff_modulus and its respective primes' bit sizes. Cells containing a "(D)" have coeff_modulus elements generated by SEAL's default function. The average was computed by taking the runtime for each sequence and dividing it by 1000, because one query was compared to 1000 documents.

Parameters	CKKS 2048 {54} (D)	CKKS 2048 {25+25}	CKKS 4096 {51 + 51}	CKKS 4096 {49 + 60}
Average encoding time:	410 μs	406 μs	870 μs	866 μs
Average encrypting time:	214 μs	298 μs	601μs	606 μs
Average ciphertext multiplication operation time:	21 μs	20 μs	39μs	38 μs
Average decrypting time:	13 μs	7 μs	14 μs	14 μs
Average relinearization time:	N/A	69 μs	144 μs	145 μ
Average decoding time:	387 μs	382 μs	831 μs	834 μs
Total average runtime:	1045 μs	1182 μs	2499 μs	2503 μs

Table 4.2

Parameters	CKKS 8192 {40 + 21 + 21 + 21 + 21 + 21 + 21 + 40}	CKKS 8192 {60 + 40 + 40 + 60}
Average encoding time:	2969 μs	2168 μs
Average encrypting time:	3183 μs	1831 μs
Average ciphertext multiplication operation time:	544 μs	234 μs
Average decrypting time:	181 μs	77 μs
Average relinearization time:	3636 μs	999 μs
Average decoding time:	3415 μs	2221 μs
Total average runtime:	13928 μs	7530 μs

Table 4.3

Parameters	CKKS 8192 {60 + 49 + 49 + 60}	CKKS 8192 {50 + 50 + 50 + 50}	CKKS 8192 {60 + 60 + 60}
Average encoding time:	2168 μs	2175 μs	2010 μs
Average encrypting time:	1833 μs	1839 μs	1533 μs
Average ciphertext multiplication operation time:	220 μs	224 μs	157 μs
Average decrypting time:	77 μs	81 μs	52 μs
Average relinearization time:	999 μs	1019 μs	601 μs
Average decoding time:	2232 μs	2262 μs	2013 μs
Total average runtime:	7529 μs	7600 μs	6366 μs

Table 4.4

Parameters	CKKS 8192 {43 + 43 + 44 + 44 + 44} (D)	CKKS 16384 {48 + 48 + 48 + 49 + 49 + 49 + 49 + 49 + 49} (D)
Average encoding time:	2340 μs	6276 μs
Average encrypting time:	2123 μs	6951 μs
Average ciphertext multiplication operation time:	308 μs	1337 μs
Average decrypting time:	103 μs	416 μs
Average relinearization time:	1480 μs	9444 μs
Average decoding time:	2470 μs	7566 μs
Total average runtime:	8824 μs	29990 μs

Table 4.5

Parameters	CKKS 16384 {60 + 40 + 40 + 40 + 60 + 60 + 60 + 60}	CKKS 16384 {60 + 31 + 31 + 60}
Average encoding time:	5915 μs	4631 μs
Average encrypting time:	6303 μs	3865 μs
Average ciphertext multiplication operation time:	1367 μs	458 μs
Average decrypting time:	363 μs	153 μs
Average relinearization time:	7635 μs	2173 μs
Average decoding time:	7030 μs	4712 μs
Total average runtime:	23686 μs	15992 μs

Table 4.6

Table 4.2 – 4.6: Benchmark results from the MicrosoftSEAL implementation without rotations, which is Procedure 5.

From the results we can see that the performance decreases as the poly_modulus_degree increases, which is to be expected. If the poly_modulus_degree increases, so does the size of the polynomial ring, which in turn also increases storage and security [35]. We can also see that more prime elements in the coeff_modulus increases runtime. A good example of this are the CKKS 8192's, starting with CKKS 8192 $\{40 + 21 + 21 + 21 + 21 + 21 + 21 + 40\}$ who has eight prime elements and the longest runtime of the CKKS 8192's with 13928 μs. Followed by CKKS 8192 $\{43+43+44+44+44\}$ with five prime elements and faster runtime of 8824 μs, but still slower than the CKKS 8192's containing four and three elements in Table 4.4. This is to be expected; recall in Section 2.8.1, how RNS seperates the polynomial computations of NTT to the different prime elements in the coeff_modulus.

Moving on to the second part of the MicrosoftSEAL benchmarks, namely the cosine similarity accuracy between decrypted ciphertexts and plaintexts. Note that when referring to cosine similarity **accuracy**, it is the percentage difference in the benchmark tables that is being shown. The Tables 4.7 and 4.8 show that the bigger the scale is, the better the accuracy is and the error has less of an impact. Additionally, it also shows that the performance between the smallest and biggest scale is practically the same and that achieving high cosine similarity accuracy does not impair performance. However, if there is no trade-off in performance, then that would open the question of checking if there is a trade-off in security. There is a connection and dependency between the poly_modulus_degree and the coeff_modulus in regards to the security of the scheme which will be discussed in evaluation Section 5.1.1.

CKKS 8192 {60 + 40 + 40 + 60}								
Scale Δ ((2^\Delta))	25	30	35	40	45	50	55	60
Average difference (%):	0.00468%	0.00019%	10^{-6}· 3.5%	10^{-7}· 2.2%	10^{-9}· 5.1%	10^{-10}· 1.2%	10^{-12}· 3.7%	10^{-13}· 1.8%
Average total time (μs):	7605 μs	-	-	-	-	-	-	7583 μs

Table 4.7

CKKS 8192 {40 + 21 + 21 + 21 + 21 + 21 + 21 + 40}								
Scale Δ (2^{Δ})	25	30	35	40	45	50	55	60
Average difference (%):	0.00665%	0.00013%	10^{-6}· 4.8%	10^{-7}· 1.4%	10^{-9}· 3.3%	10^{-10}· 2.7%	10^{-12}· 4.5%	10^{-13}· 2.4%
Average total runtime (μs):	13683 μs	-	-	-	-	-	-	13789 μs

Table 4.8

Table 4.7 – 4.8: Benchmark results over different scales of Δ from comparing cosine similarity between decrypted ciphertexts and plaintexts.

4.3 Benchmark using rotation and rescaling

This section showcases the benchmarks for Procedure 6, when rotation is implemented in order to perform addition of all the elements in the ciphertext. Additionally, performing rotation on low parameters, such as CKKS 2048 {54}, was not possible in MicrosoftSEAL because it required more than one prime element in the coeff_modulus. This section will also showcase the performance of rescaling, shown in Table 4.9. Certain parameters gain performance from rescaling, it speeds up rotation and relinearization. However, rescaling can not be performed on all parameters, these have the result of "N/A", where rescaling was not possible to implement due to short modulus chain from the coeff_modulus. Moreover, rotation operations that were performed after rescaling are noted with **(rescaled)**. The last column in Table 4.9 presents the amount of time saved from performing rescaling before rotation.

Parameters	Average rotation time:		Average rescaling time:	Time saved:
CKKS 2048 {25+25}	790 μs		N/A	-
CKKS 4096 {51+51}	1670 μs		N/A	-
CKKS 4096 {49+60}	1649 μs		N/A	-
CKKS 8192 {43+43+44+44+44}	16023 μs	(rescaled) 10838 μs	394 μs	4791 μs
CKKS 8192 {60+40+40+60}	10964 μs	(rescaled) 6749 μs	295 μs	3920 μs
CKKS 8192 {60+49+49+60}	10845 μs	(rescaled) 6633 μs	293 μs	3919 μs
CKKS 8192 {50+50+50+50}	10900 μs	(rescaled) 6604 μs	295 μs	4001 μs
CKKS 8192 {60+60+60}	6661 μs	(rescaled) 3568 μs	207 μs	2886 μs
CKKS 8192 {40+21+21+21+21+21+21+40}	38380 μs	(rescaled) 29844 μs	711 μs	7825 μs
CKKS 16384 {48+48+48+49+49+49+49+49+49}	100885 μs	(rescaled) 81797 μs	1723 μs	17365 μs
CKKS 16384 {60+40+40+40+60+60+60+60}	83416 μs	(rescaled) 64317 μs	1524 μs	17575 μs
CKKS 16384 {60+31+31+60}	23329 μs	(rescaled) 14169 μs	627 μs	8533 μs

Table 4.9: Performance benchmark of average rescaling time, performance benchmark of rotation with and without rescaling, and time saved from using rescaling before rotation.

It is worth to note that, modulus chains that allow for more than one rescale, for example {60+21+21+21+60} with a scale of 2^{60} can drop all three 21-bitsized intermediate primes, which will further decrease rotation time. In Figure 4.10 and 4.11 we see the benchmarks for parameters CKKS 8192 {60+40+40+60} and CKKS 16384 {60+31+31+60}, which includes the time saved for performing two rescales compared to just once. It also shows that with a second rescale the accuracy of the decrypted cosine similarity is worse than one rescale. Furthermore, in Figure 4.12 we showcase that performing rescale before relinearization, instead of after relinearization, also improves the overall performance by a couple hundred microseconds. However, for larger poly_modulus_degrees the impact is doubled or quadrupled.

CKKS 8192 {60+40+40+60}	Rotation time:	Rescale time:	Relinearization time:	Total time and difference:		Cosine similarity difference:
One rescale	6631 μs	435 μs	603 μs	7669 μs	3306 μs	$10^{-12} \cdot 4.596\%$
Two rescales	3342 μs	719 μs	302 μs	4363 μs		0.00223%

Table 4.10: Benchmark comparison of performing one or two rescaling, for parameter CKKS 8192 {60+40+40+60}

CKKS 16384 {60+31+31+60}	Rotation time:	Rescale time:	Relinearization time:	Total time and difference:		Cosine similarity difference:
One rescale	14170 μs	941 μs	1303 μs	16561 μs	7021 μs	$10^{-12} \cdot 3.908\%$
Two rescales	7157 μs	1577 μs	659 μs	9393 μs		$10^{-8} \cdot 5.748\%$

Table 4.11: Benchmark comparison of performing one or two rescaling, for parameter CKKS 16384 {60+31+31+60}

Parameters	Rescaling after relinearization:		Rescaling before relinearization:		Time saved:
	Rescaling time:	Relinearization time:	Rescaling time:	Relinearization time:	
CKKS 8192 {43+43+44+44+44}	394 μs	1486 μs	598 μs	1014 μs	268 μs
CKKS 8192 {60+40+40+60}	295 μs	1022 μs	445 μs	657 μs	215 μs
CKKS 8192 {60+49+49+60}	293 μs	1000 μs	443 μs	615 μs	235 μs
CKKS 8192 {50+50+50+50}	295 μs	997 μs	447 μs	611 μs	234 μs
CKKS 8192 {60+60+60}	207 μs	638 μs	305 μs	314 μs	226 μs
CKKS 8192 {40+21+21+21+21+21+21+40}	711 μs	3631 μs	1078 μs	2871 μs	393 μs
CKKS 16384 {48+48+48+49+49+49+49+49}	1723 μs	9439 μs	2565 μs	7540 μs	1057 μs
CKKS 16384 {60+40+40+40+60+60+60+60}	1524 μs	7561 μs	2280 μs	5881 μs	924 μs
CKKS 16384 {60+31+31+60}	627 μs	2173 μs	943 μs	1304 μs	553 μs

Table 4.12: Benchmark comparison of performing rescaling before relinearization

4.4 Benchmark plaintext documents

The benchmark results for plaintext-ciphertext computations will be presented in this section. In Table 4.13 the runtime for the different sequences is shown. Note that the encryption is only performed on the queries, hence the **(queries)** note in the table. Additionally, there is no average relinearization time, because relinearization is not required when plaintext-ciphertext computations are executed. What we can also see from this result is that the plaintext-ciphertext multiplication is less than half the runtime of plaintext-ciphertext multiplication from previous benchmarks like in Table 4.3.

Parameters	CKKS 8192 {40 + 21 + 21 + 21 + 21 + 21 + 21 + 40}	CKKS 8192 {60 + 40 + 40 + 60}
Average encoding time:	2854 μs	2173 μs
Average encrypting time **(queries)**:	3079 μs	1830 μs
Average ciphertext multiplication operation time **(plaintext)**:	263 μs	110 μs
Average ciphertext multiplication operation time **(ciphertext)**:	544 μs	234 μs
Average decrypting time:	179 μs	77 μs
Average decoding time:	3379 μs	2202 μs

Table 4.13: Results from benchmarks using plaintext documents

Tables 4.14 and 4.15 show that for scales 30, 50 and 60 there is a disparity compared to the ciphertext-ciphertext computations from Tables 4.7 and 4.8. In Table 4.15 the average accuracy for scale 30 has been halved, thus moving down one decimal point in accuracy. Same thing goes for scale 50 and 60 in both Table 4.14 and Table 4.15, which went from 10^{-10} to 10^{-11} and 10^{-13} to 10^{-14} respectively.

CKKS 8192 {60 + 40 + 40 + 60}								
Scale Δ ((2^\Delta))	25	30	35	40	45	50	55	60
Average difference (%):	0.00194%	0.00010%	$10^{-6} \cdot$ 6.6%	$10^{-7} \cdot$ 1.2%	$10^{-9} \cdot$ 1.3%	$10^{-11} \cdot$ 5.1%	$10^{-12} \cdot$ 5.6%	$10^{-14} \cdot$ 9.1%
Average total time (μs):	6392 μs	-	-	-	-	-	-	6410 μs

Table 4.14

CKKS 8192 {40 + 21 + 21 + 21 + 21 + 21 + 40}								
Scale Δ (2^{Δ})	25	30	35	40	45	50	55	60
Average difference (%):	0.00226%	0.00005%	$10^{-6}\cdot$ 1.7%	$10^{-7}\cdot$ 1.9%	$10^{-9}\cdot$ 3.6%	$10^{-11}\cdot$ 6.9%	$10^{-12}\cdot$ 2.6%	$10^{-14}\cdot$ 8.2%
Average total runtime (μs):	9873 μs	-	-	-	-	-	-	9754 μs

Table 4.15

Table 4.11 – 4.12: Benchmark results over different scales of Δ from comparing cosine similarity between plaintext and decrypted plaintext-ciphertext-multiplication ciphertexts.

4.5 Database using LSH

The embedding model we used was all-MiniLM-L6-v2 which outputs 384-dimensional vectors. By using 384 random projections of 384 dimensions we were able to get the benchmarks shown below of the LSH implementation. For each query we compare the amount of documents that LSH retrieved with the vectors that our plaintext retrieval obtained. The plaintext retrieval was performed using Weaviate [40] querying for the cosine similarity. Weaviate allows for a flat search which guarantees the optimal result because it conducts a linear search through all stored embeddings. We decided on using Weaviate because of familiarity, flat search functionality and it being open-source.

The diagrams below are the results from iterating over 500 query embeddings against the entire set of document embeddings from the SCIDOCS dataset which has 25637 documents. The diagrams show the overlap between the LSH retrieval and the optimal retrieval returned by Weaviate (for retrieved document counts 10,25 and 50).

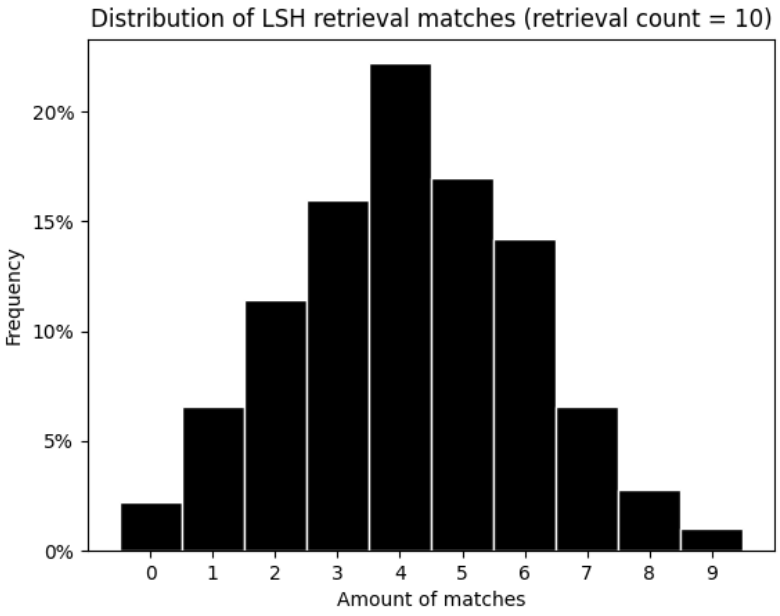

Figure 4.1: Amount of matches when retrieving 10 documents using LSH from database compared to 10 documents from Weaviate.

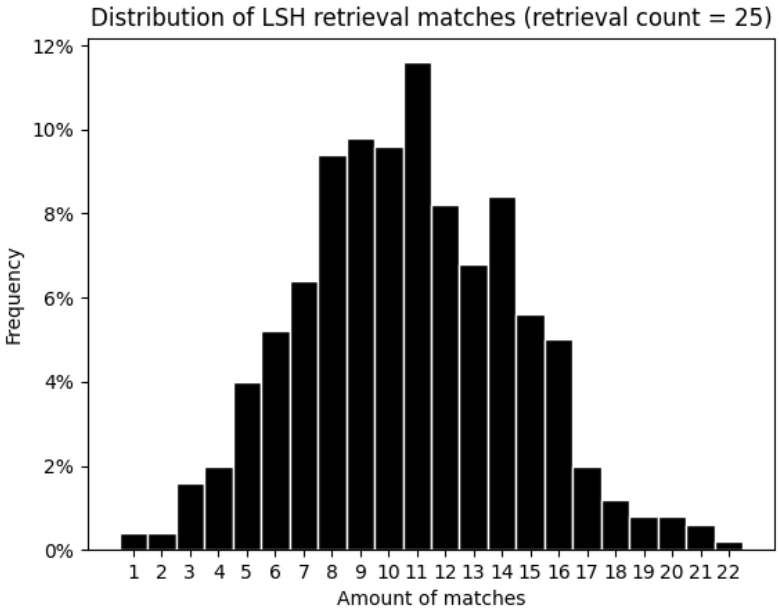

Figure 4.2: Amount of matches when retrieving 25 documents using LSH from database compared to 25 documents from Weaviate.

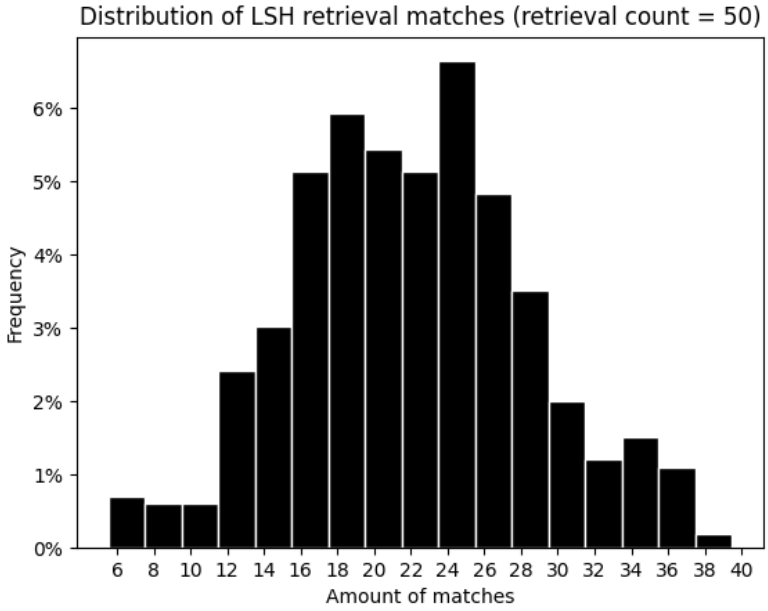

Figure 4.3: Amount of matches when retrieving 50 documents using LSH from database compared to 50 documents from Weaviate.

Below is a table that describes these results in another format to aid the reader in understanding the retrieval results of the LSH implementation. The table shows *mean, median, mode, max* and *min* for each retrieval count that has been tested.

We see a quite wide range of matches, however the *mean* is centered around 41.4%, 42.8% and 44.3% for the respective retrieval counts 10, 25 and 50.

Retrieval Count	**Mean**	**Median**	**Mode**	**Min**	**Max**
10	4.14	4	4	0	9
25	10.692	11	11	1	22
50	22.148	22	24	6	40

Table 4.16: Table that depicts LSH retrieval benchmarks result in another format by showing mean, median, mode, max and min for each retrieval count tested.

We see that the retrieval can be improved by increasing the limit for the amount of rows retrieved by the LSH fingerprint lookup. However, the more we increase this limit the more homomorphic operations are needed on the server, which will consequently linearly increase the size of the response sent to the client. In Section 5.1.3 we discuss other improvements that should be considered first.

5

Discussion

This thesis aimed to investigate the use of HE within cloud storage. The problem we found was that private vector similarity search on the cloud, where a user could conduct semantic search on textual data using word embeddings and retrieving relevant stored data without divulging any information about the query nor the stored data, has not yet been fully explored. In Section 5.1 we evaluate our results and present our interpretation of them. As we progress, we also discuss nuances and present results where we chose not to enforce the hard rule of end-to-end privacy. This encapsulates the security- and privacy preserving limitations. Furthermore, we present a future works section that includes comparative performance of homomorphic multiplication in CKKS on GPUs, a distributed HE scheme and an alternate cryptographic solution that does not use HE. We end the discussion section with our conclusion.

5.1 Evaluation

To reason about the efficacy about HE in the use-case of similarity search we must first create a distinction of what happens on the client and what happens on the server. The server will handle database storage and retrieval as well as performing the homomorphic operations that result in the similarity measure that the user is querying for. The client will on the other hand prepare documents and queries to be stored or operated on respectively. In Table 5.1 we show the amount of cosine similarities that the server can execute in one second; this is without accounting for database retrieval latency since we have not optimized indexing on the database and thus not measured retrieval time. Additionally, it is possible to offload HE operations to the GPU which makes database retrieval less relevant since the two will not be interfering with each other for CPU time as often.

- Method 1, the TenSEAL benchmarks, Section 4.1,
- Method 2, the MicrosoftSEAL benchmarks without rotation, Section 4.2.
- Method 3, the MicrosoftSEAL benchmarks with rotation, Section 4.3.
- Method 4, the MicrosoftSEAL plaintext documents benchmarks without rotation, Section 4.4.
- Method 5, the MicrosoftSEAL plaintext documents benchmarks with rotation,

combining Sections 4.3 and 4.4.

Method #	Method 1	Method 2	Method 3	Method 4	Method 5
1s	64 cosine similarities	811 cosine similarities	217 cosine similarities	9090 cosine similarities	253 cosine similarities

Table 5.1: Number of cosine similarities computed in one second for Methods 1 to 5 with parameters CKKS 8192 {60+40+40+60}

The results in Table 5.1 were calculated by taking the corresponding runtimes from Section 4. For Method 1 we took the computation time from Table 4.1. For Method 2 we took the average ciphertext multiplication operation time and added it together with the average relinearization time. For Method 4 we only took the average ciphertext multiplication operation time for plaintext documents, because there is no relinearization, rotation or rescaling. For Methods 3 and 5 we take the fastest total time from Figure 4.10 and add the multiplication operation time. Note that for Method 5 will subtract the relinearization operation time and also subtract half of the multiplication operation time. The times taken from each table all correspond to the parameter CKKS 8192 {60+40+40+60}.

5.1.1 Preserving Privacy & Security

Section 4 presented four different tables where the cosine similarity was benchmarked on scales 25 to 60. What was quickly deduced from these tables is that the scale does not impact performance. However, if increasing the scale to achieve the highest possible accuracy does not impact performance, then increasing the scale must impact security. Albrecht et al. provide tables of recommended parameters, shown in Table 5.2, which will be used to estimate the parameters' impact to security.

Distribution (-1,1)									
poly_modulus_degree	1024			2048			4096		
bit security	128	192	256	128	192	256	128	192	256
log q	27	19	14	54	37	29	109	75	58
poly_modulus_degree	8192			16384			32768		
bit security	128	192	256	128	192	256	128	192	256
log q	218	152	118	438	305	237	881	611	476

Table 5.2: Part of the tables of recommended parameters by Albrecht et al. [35].

The parameter **log q** from Table 5.2 refers to the multiplication of all the prime elements' bit size in the coeff_modulus. Take parameter CKKS 8192 {60+40+40+60} for example: Go to poly_modulus_degree 8192, take the sum of the coeff_modulus which is 200, as long as the sum from the coeff_modulus elements is less than the value of **log q**, then that parameter has that level of bit security, in this case that would be 128-bit security. Therefore, if CKKS 8192 needs 256-bit security then

achieving a scale of 60 would not be possible. Recall in Section 2.8.1 how rescaling requires the sum of the primes' bit size to be equal or higher than the scale used to multiply two ciphertexts together. If the maximum total bit size of the prime elements in the coeff_modulus is 118, then we hypothesize that the most optimal coeff_modulus is {41+35+41}, which limits the scale Δ to $37 < (38 = ((41+35)/2))$.

Moving on to the security of AES. The security of AES is established and its security depends on the AES scheme's bit-size, whether it is 128, 192 or 256-bits which all have a bit security corresponding to their bit size (i.e. AES-256 has a bit security of 256). AES is also shown to be post quantum resistant which means that it does not pose more risk in our implementation, assuming the key is safely stored.

The reason we decided on encrypting word embeddings even though some mitigations exist outside the cryptographic domain was to create a robust and timeless solution for the problem. One protection technique proposed by the paper by Zhuang et al. presented a mitigation that relies on random transformations of embeddings to prevent inversion [16]. This technique is security through obscurity, which is not secure. Furthermore, there is no way to guarantee that future methods will not be able to bypass the aforementioned mitigations in Section 2.2.1. Additionally, internal malicious actors could have black-box access to the embedding models, which we must account for. Thus, we believe that implementing a cryptographically secure solution for the protection of word embeddings is the more robust way. Our main subject was Homomorphic Encryption however we discuss a symmetric cryptography solution that is adjacent to our proposed solution in Section 5.2.3. On the other hand it is possible to securely host everything locally, an organization could secure their network and only allow internal access to the database. This would require knowledge of network security and was not the subject of our thesis hence was not something we have discussed.

5.1.2 Comparison of private and public data

As stated in Section 1.2, our objective is to ensure that both the documents stored in the cloud and the queries sent from the client remain encrypted and private at all times. Encrypting stored documents protects user privacy in the event of a data breach or unauthorized access. Similarly, encrypting client queries prevents the risk of information leakage due to interception during transmission, as an adversaries could exploit unencrypted queries to reconstruct sensitive user data through inversion attacks. However, end-to-end encryption increases both the execution time of semantic similarity computations and the overhead for communication between the server and client. Therefore, a comparison in which one dataset was public was necessary to evaluate the associated trade-offs.

As mentioned in Section 3.5, Kim et al. [39] gave the main idea to perform this comparison. However, their paper focuses on preventing inversion attacks on queries with homomorphic encryption (CKKS) and noise-based perturbation. Whilst our report is not mainly about inversion attacks, they are intrinsically a part of any work that uses or is related to word embeddings and its security. Still, they do compare the baseline plaintext queries' cosine similarity with homomorphically encrypted

queries' cosine similarity and also with queries who have been altered with something known as d_χ-privacy, which is a form of perturbation. It comparatively shows that the cosine similarity for the perturbated plaintexts gets increasingly worse as the noise and privacy increases. Interestingly enough, the homomorphically encrypted queries' cosine similarity seem to be in line with our benchmark results and perform very well. Their result is within the percentage difference of 10^{-4} to 10^{-6}, which is consistent with certain parameters in our results as well.

With the context of the paper by Kim et al. [39] explained, we can move on to the impact of our benchmark results. Recall that with plaintext-ciphertext multiplication, the resulting ciphertext does not increase in size and therefore no relinearization operation is required. This saves computational runtime for the server, and in the case of Table 5.1 it would provide more cosine similarities. Therefore, if the user decided that they do not need both queries and documents to be encrypted, but only one, they could increase the amount of computations by around 16-17%.

5.1.3 Database using LSH

The benchmark results of the LSH retrieval seem satisfactory, the distribution is centered just above 40% matches given a retrieval count r. However we believe that this benchmark can be improved upon. By decrypting the similarity measure and sorting the list of retrieved documents we would also be able to see the order of the matched documents. When retrieving 10 documents, assume our LSH retrieval has 5 matches with the plaintext lookup. We see the need to explore the distribution of these 5 matches: Are they evenly distributed from best to worst similarity? Are they usually biased to being most relevant documents or least relevant documents? This fine grained benchmark improvement would give a more complete view of the results.

What we faced was the issue of efficient retrieval since it would be infeasible to compute similarity metrics between the query CKKS vector and all stored CKKS vectors. This would lead to a significant computation time as well as a query response that would grow linearly along with the amount of stored documents. This would make the implementation seem very infeasible and unnecessary.

Some ways we believe this metric can be improved is through fine-tuning of embedding models or a dataset dependent set of LSH bases. We believe that a combination of these two would provide for a lot better LSH based retrieval. The fine tuning route would specifically mitigate what can be described as a narrow cone of embeddings when embedding topically related sentences. The fine-tuning would essentially increase the span of the cone making cosine similarities and LSH fingerprints more accurate [41]. This would also make our LSH fingerprints more accurate since the space our vectors occupy will become more sparse. Another way is to test other embedding models and see if they encode embeddings in a broader way, this would result in higher amount of optimal retrieval by LSH. The other way is to choose the bases not randomly but dependent on the dataset. However this introduces issues when new data is added to the dataset, because the bases must be recomputed. It is important to note that the Hyperplanes LSH has been shown to outperform

data-dependent LSH algorithms by adding more bases as stated in Section 2.4.

5.2 Future works

In this section, potential real-world applications and adaptations from a technical report and our findings will be discussed.

5.2.1 Comparison of performance and real use-case

While there are not any exact comparable benchmarks to be used for our evaluation, there are some performance tests that we can extrapolate and use for our use-case and argue its viability. One such performance test comes from a paper by Li and Zong [42]. Their results compare the speed of several homomorphic operations between an AMD EPYC 9654 CPU, an Nvidia 3090 GPU and an Nvidia 4090 GPU. Most notably they present a **356x** speedup for ciphertext-ciphertext multiplication, a speedup between the AMD EPYC 9654 CPU and the Nvidia 4090 GPU. Li and Zong also present a **1687x** speedup for plaintext-ciphertext multiplication. Additionally, our Apple Silicon M3 chip compared to the AMD EPYC 9654 CPU have close enough performance to not require any re-calculations.

Moreover, to answer one of our research questions: "Is it viable enough to be used in the real-world for our specific use-case?". It is our understanding of modern cloud providers and the current standard of hardware for such servers that, this would be a viable option for semantic similarity search. The only caveat would be that the implementation must be extended to support multiple clients querying the same dataset, which is discussed in the next section.

5.2.2 Distributed HE

One of the future research questions that could be explored and tested, is the use of HE by multiple parties. A distributed HE is an idea that Albrecht et al. [35] presented, they then established a distributed-key-generation (DKG) algorithm and its input parameters, number of parties t, and threshold parameter d. DKG's output is a secret key $s = \{s_1, \ldots, s_t\}$ where d amount of parties are needed for decryption. Unlike normal HE, where one would simply use the public key to encrypt a message M, Albrecht et al. have described a distributed-encryption (DE) algorithm that takes a message M and the secret key s_i from a party. The output is a ciphertext C which can get decrypted using the distributed-decryption (DD) algorithm which also takes as input the threshhold d and a subset of secret keys $s = \{s_1, \ldots, s_t\}$.

However, there is no explicit explanation of how the DE and DD algorithms work in the technical report, therefore we provide our protocol procedure as following: We dismiss the DE algorithm and simply use the public key to encrypt messages to then be stored on the cloud. Additionally, because each party has a part of the secret key, each party can perform a partial decryption of the ciphertext $c_1 \cdot s_i$. This will start a collaboration, where each participating party needs to be verified. Verification could be performed through (H)MACs or zero-knowledge proofs. Once the client

has received $C = \{c_1 \cdot s_1, ..., c_1 \cdot s_k\}$, where k is the degree of the polynomial plus one, the message can now be decrypted using Lagrange interpolation without revealing the secret key.

Albrecht et al. gave an example where multiple parties had to sign off on the cooperation, however what if a single party wanted access to the encrypted data and only needed to be verified. An example would be for a department at a university implementing a RAG for scientific papers and a student wanting to query these encrypted scientific papers from the university's servers. The student would first have to verify that they are a student, afterwards Shamir's Secret Sharing scheme would execute and the student would receive the secret key to decrypt the documents. Instead of using the DD algorithm, the client would retrieve k-parts of the secret key to then rebuild the whole secret key, which is illustrated in Figure 5.1. However, there is a problem with this idea. This would mean that the secret key would be revealed to a new party, which is often avoided in cryptography. In most privacy-preserving or secure multi-party applications, you want to prevent any party from ever learning the secret key. Lastly, the question still remains of how the rebuilt secret key should be safely discarded after it has been used by the client.

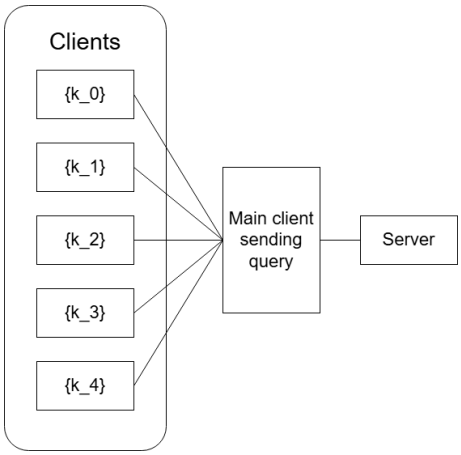

Figure 5.1: Illustration of the Shamir's secret sharing scheme for a client querying

5.2.3 Alternate solution using AES only

The question of "How much can be offloaded to the server versus client?" is an interesting question to think about, we find our implementation to foster both paradigms. One of the backbones of our architecture is an efficient ANN search with LSH, from which one can easily derive a client-only computation scenario. The server would only be used for database storage and retrieval. In that case, we could store plaintext vectors encrypted only using AES and compute the similarity measures solely on the client after server retrieval using LSH. This would essentially allow the server to serve vastly more clients without throttling requests. If the user wants to offload as much computation as possible to the server our HE architecture coupled with LSH provides a good first step in that direction.

5.3 Conclusion

Given the conducted experiments, benchmark results and LSH implementation, the conclusive assumption is that a performant and secure HE based vector database is feasible. This assumption is also partly based on the performance of an Nvidia 4090 GPU [42]. Additionally, to achieve the highest accuracy with a scale Δ of 60, faster runtime with two rescaling operations, and to also achieve a 256-bit security, the minimum required parameters for the best performance is: CKKS 16384 {60+31+31+60}. This will perform 101 cosine similarity computations on the server in one second. With the performance of the Nvidia 4090 GPU we conclude that there is still more runtime performance to be gained compared to the performance benchmarks we have presente. Taking into account the **356x** speedup for ciphertext-ciphertext multiplication from Li and Zong [42], leads us to claim that a server cluster with 8 different hosts could compute approximately $101 \cdot 356 \cdot 8 = 287\ 648$ cosine similarities per second. Given that database retrieval is not a GPU operation, it will not throttle the performance of running the HE operations on the GPU. Note that, we omit any CPU interference between HE operations on the GPU and database operations for simplicity. Furthermore, if we perform plaintext-ciphertext multiplication, meaning either the query or the document is not encrypted, then we get a total of 119 cosine similarity computations on the server in one second. With the same server cluster of 8 different hosts containing Nvidia 4090 GPUs, and taking into account the **1687x** speedup from Li and Zong [42], we get approximately $119 \cdot 1687 \cdot 8 = 1\ 606\ 024$ cosine similarities per second.

Bibliography

[1] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. P. Fitzek, and N. Aaraj, "Survey on fully homomorphic encryption, theory, and applications," *Proceedings of the IEEE*, vol. 110, no. 10, pp. 1572–1609, 2022. DOI: 10.1109/JPROC.2022.3205665.

[2] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Comput. Surv.*, vol. 51, no. 4, Jul. 2018, ISSN: 0360-0300. DOI: 10.1145/3214303. [Online]. Available: https://doi.org/10.1145/3214303.

[3] C. Bolgar, *Microsoft's majorana 1 chip carves new path for quantum computing*, Accessed: 2025-03-10, Feb. 2025. [Online]. Available: https://news.microsoft.com/source/features/innovation/microsofts-majorana-1-chip-carves-new-path-for-quantum-computing/.

[4] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *J. ACM*, vol. 56, no. 6, Sep. 2009, ISSN: 0004-5411. DOI: 10.1145/1568318.1568324. [Online]. Available: https://doi.org/10.1145/1568318.1568324.

[5] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23, ISBN: 978-3-642-13190-5.

[6] J. H. Conway and N. J. A. Sloane, *Sphere Packings, Lattices and Groups* (Grundlehren der mathematischen Wissenschaften), 2nd. New York: Springer, 1993, vol. 290, ISBN: 978-1-4757-2249-9.

[7] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, Nov. 2013, ISSN: 0004-5411. DOI: 10.1145/2535925. [Online]. Available: https://doi.org/10.1145/2535925.

[8] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds., Cham: Springer International Publishing, 2017, pp. 409–437, ISBN: 978-3-319-70694-8.

[9] T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient estimation of word representations in vector space*, 2013. arXiv: 1301.3781 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1301.3781.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2019. arXiv: 1810.04805 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1810.04805.

[11] M. Oquab, T. Darcet, T. Moutakanni, *et al.*, *Dinov2: Learning robust visual features without supervision*, 2024. arXiv: 2304.07193 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2304.07193.

[12] A. Radford, J. W. Kim, C. Hallacy, *et al.*, *Learning transferable visual models from natural language supervision*, 2021. arXiv: 2103.00020 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2103.00020.

[13] A. Baevski, H. Zhou, A. Mohamed, and M. Auli, *Wav2vec 2.0: A framework for self-supervised learning of speech representations*, 2020. arXiv: 2006.11477 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2006.11477.

[14] C. Song and A. Raghunathan, "Information leakage in embedding models," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 377–390, ISBN: 9781450370899. DOI: 10.1145/3372297.3417270. [Online]. Available: https://doi.org/10.1145/3372297.3417270.

[15] J. X. Morris, V. Kuleshov, V. Shmatikov, and A. M. Rush, *Text embeddings reveal (almost) as much as text*, 2023. arXiv: 2310.06816 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2310.06816.

[16] S. Zhuang, B. Koopman, X. Chu, and G. Zuccon, *Understanding and mitigating the threat of vec2text to dense retrieval systems*, 2024. arXiv: 2402.12784 [cs.IR]. [Online]. Available: https://arxiv.org/abs/2402.12784.

[17] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02, Montreal, Quebec, Canada: Association for Computing Machinery, 2002, pp. 380–388, ISBN: 1581134959. DOI: 10.1145/509907.509965. [Online]. Available: https://doi.org/10.1145/509907.509965.

[18] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, ser. SCG '04, Brooklyn, New York, USA: Association for Computing Machinery, 2004, pp. 253–262, ISBN: 1581138857. DOI: 10.1145/997817.997857. [Online]. Available: https://doi.org/10.1145/997817.997857.

[19] D. Cai, *A revisit of hashing algorithms for approximate nearest neighbor search*, 2019. arXiv: 1612.07545 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1612.07545.

[20] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, *Practical and optimal lsh for angular distance*, 2015. arXiv: 1509.02897 [cs.DS]. [Online]. Available: https://arxiv.org/abs/1509.02897.

[21] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07, Vienna, Austria: VLDB Endowment, 2007, pp. 950–961, ISBN: 9781595936493.

[22] Y. Lin, D. Cai, and C. Li, *Density sensitive hashing*, 2012. arXiv: 1205.2930 [cs.IR]. [Online]. Available: https://arxiv.org/abs/1205.2930.

[23] A. Andoni and I. Razenshteyn, *Optimal data-dependent hashing for approximate near neighbors*, 2015. arXiv: 1501.01062 [cs.DS]. [Online]. Available: https://arxiv.org/abs/1501.01062.

[24] P. Lewis, E. Perez, A. Piktus, *et al.*, *Retrieval-augmented generation for knowledge-intensive nlp tasks*, 2021. arXiv: 2005.11401 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2005.11401.

[25] Microsoft Research, *Microsoft SEAL (Simple Encrypted Arithmetic Library)*, Accessed: 2024-11-30, 2019. [Online]. Available: https://www.microsoft.com/en-us/research/project/microsoft-seal/.

[26] J. Fan and F. Vercauteren, *Somewhat practical fully homomorphic encryption*, Cryptology ePrint Archive, Paper 2012/144, 2012. [Online]. Available: https://eprint.iacr.org/2012/144.

[27] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12, Cambridge, Massachusetts: Association for Computing Machinery, 2012, pp. 309–325, ISBN: 9781450311151. DOI: 10.1145/2090236.2090262. [Online]. Available: https://doi.org/10.1145/2090236.2090262.

[28] A. Benaissa, B. Retiat, B. Cebere, and A. E. Belfedhal, *Tenseal: A library for encrypted tensor operations using homomorphic encryption*, 2021. arXiv: 2104.03152 [cs.CR].

[29] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, Nov. 2019. [Online]. Available: https://arxiv.org/abs/1908.10084.

[30] H. Chen, K. Han, Z. Huang, A. Jalali, and K. Laine, "Simple encrypted arithmetic library v2.3.0," 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:53649014.

[31] J.-C. Bajard, J. Eynard, A. Hasan, and V. Zucca, *A full RNS variant of FV like somewhat homomorphic encryption schemes*, Cryptology ePrint Archive, Paper 2016/510, 2016. [Online]. Available: https://eprint.iacr.org/2016/510.

[32] A. Satriawan, R. Mareta, and H. Lee, *A complete beginner guide to the number theoretic transform (NTT)*, Cryptology ePrint Archive, Paper 2024/585, 2024. DOI: 10.1109/ACCESS.2023.3294446. [Online]. Available: https://eprint.iacr.org/2024/585.

[33] *Microsoft SEAL (release 4.1)*, https://github.com/Microsoft/SEAL, Microsoft Research, Redmond, WA., Jan. 2023.

[34] M. Blatt, A. Gusev, Y. Polyakov, K. Rohloff, and V. Vaikuntanathan, "Optimized homomorphic encryption solution for secure genome-wide association studies," *BMC Medical Genomics*, vol. 13, no. 7, p. 83, 2020, ISSN: 1755-8794. DOI: 10.1186/s12920-020-0719-9. [Online]. Available: https://doi.org/10.1186/s12920-020-0719-9.

[35] M. Albrecht, M. Chase, H. Chen, *et al.*, "Homomorphic encryption security standard," HomomorphicEncryption.org, Toronto, Canada, Tech. Rep., Nov. 2018.

[36] H. Chen, K. Han, Z. Huang, A. Jalali, and K. Laine, "Simple encrypted arithmetic library v2.3.0," 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:53649014.

[37] N. Thakur, N. Reimers, A. Rücklé, A. Srivastava, and I. Gurevych, "BEIR: A heterogeneous benchmark for zero-shot evaluation of information retrieval models," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. [Online]. Available: https://openreview.net/forum?id=wCu6T5xFjeJ.

[38] S. Serengil and A. Ozpinar, *Cipherface: A fully homomorphic encryption-driven framework for secure cloud-based facial recognition*, 2025. arXiv: 2502.18514 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2502.18514.

[39] D. Kim, G. Lee, and S. Oh, "Toward privacy-preserving text embedding similarity with homomorphic encryption," in *Proceedings of the Fourth Workshop on Financial Technology and Natural Language Processing (FinNLP)*, C.-C. Chen, H.-H. Huang, H. Takamura, and H.-H. Chen, Eds., Abu Dhabi, United Arab Emirates (Hybrid): Association for Computational Linguistics, Dec. 2022, pp. 25–36. DOI: 10.18653/v1/2022.finnlp-1.4. [Online]. Available: https://aclanthology.org/2022.finnlp-1.4/.

[40] E. Dilocker, B. van Luijt, B. Voorbach, *et al.*, *Weaviate*. [Online]. Available: https://github.com/weaviate/weaviate.

[41] T. Gao, X. Yao, and D. Chen, *Simcse: Simple contrastive learning of sentence embeddings*, 2022. arXiv: 2104.08821 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2104.08821.

[42] Q. Li and R. Zong, *Cat: A gpu-accelerated fhe framework with its application to high-precision private dataset query*, 2025. arXiv: 2503.22227 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2503.22227.

A

Appendix 1

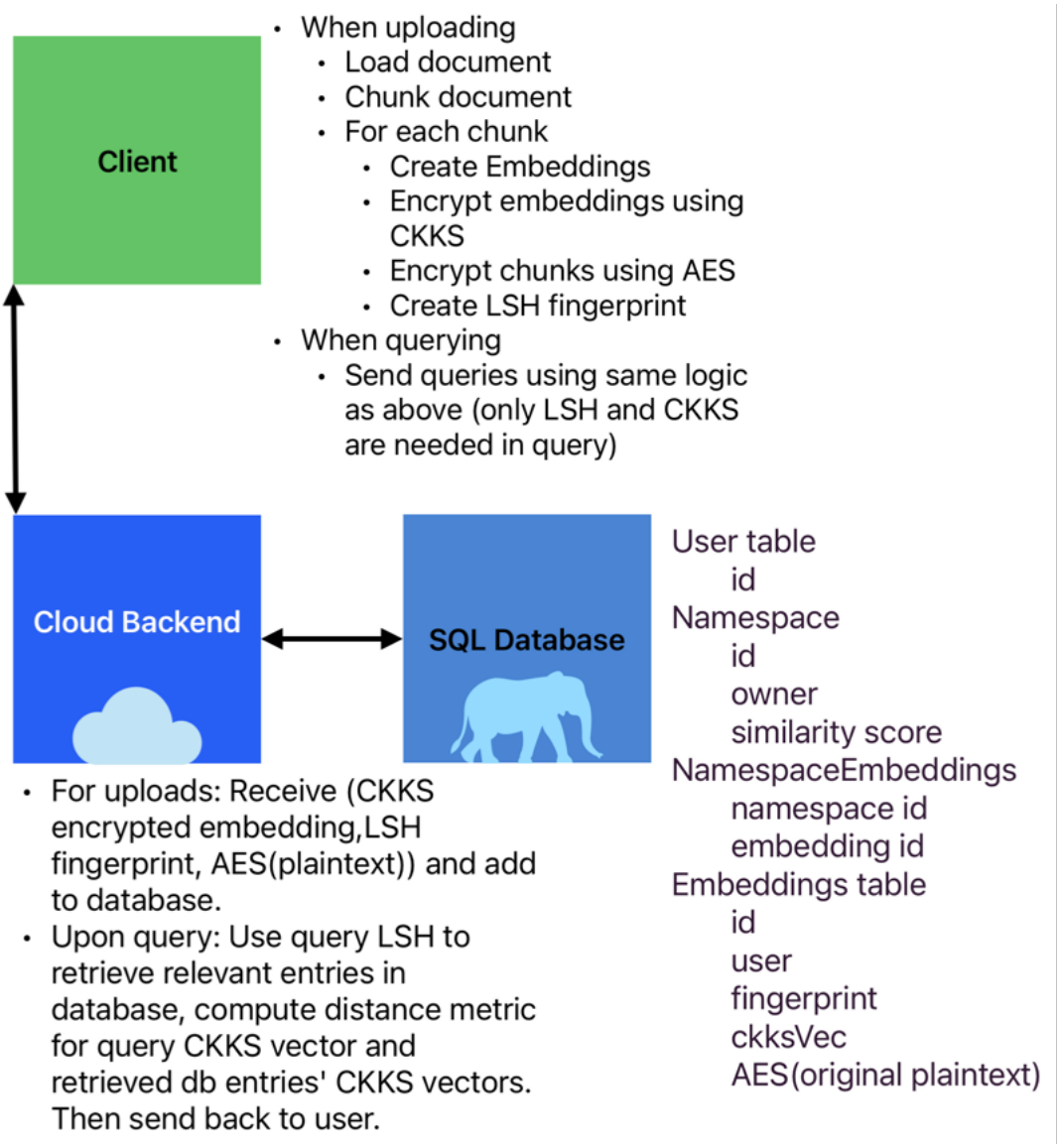

Figure A.1: Prototype architecture

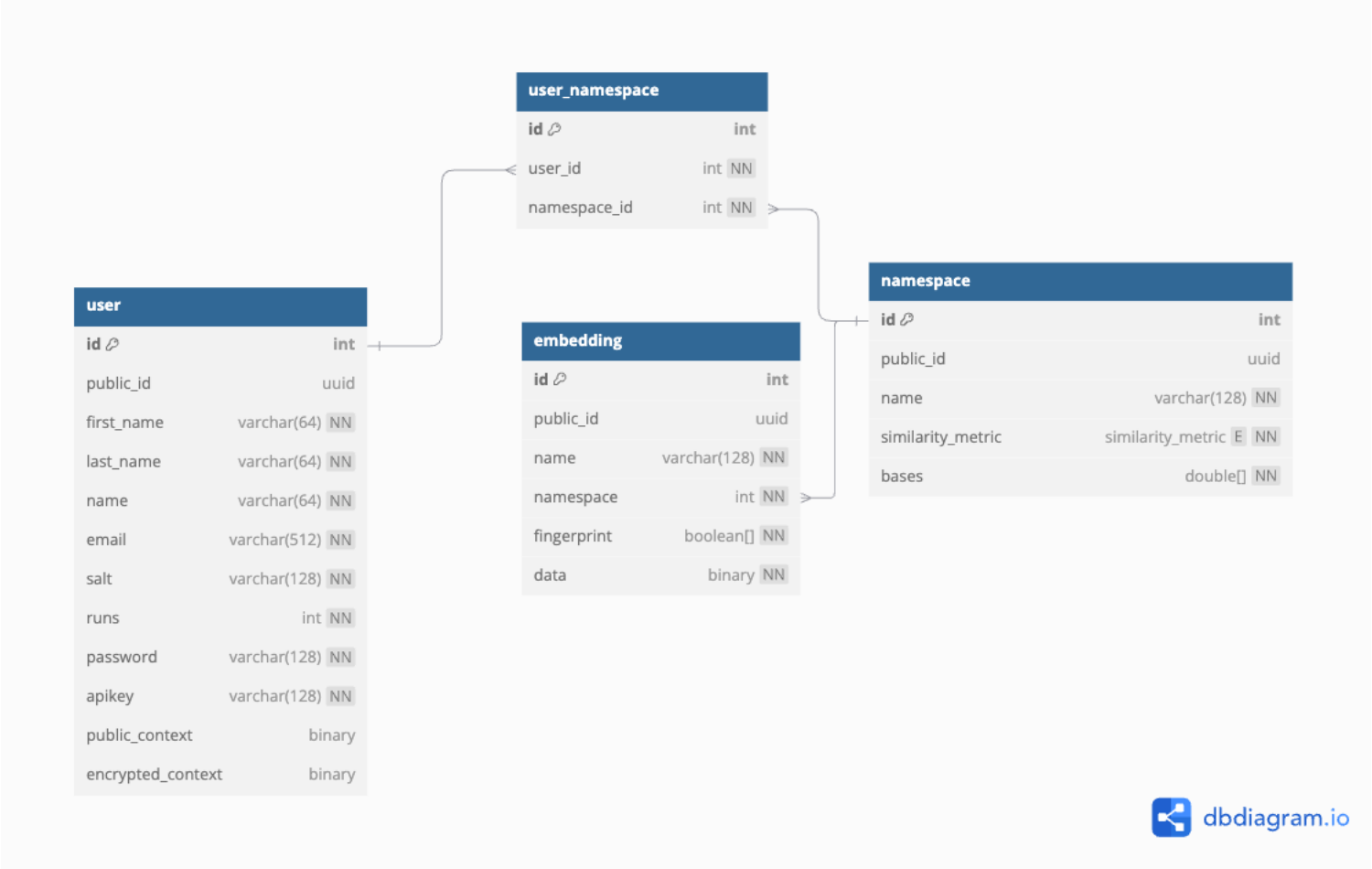

Figure A.2: Database schema from the client and server prototype