Evaluation des praktischen Nutzen von homomorpher Verschlüsselung

Projektarbeit T2000

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik / Informationstechnik

an der

Duale Hochschule Baden-Württemberg Karlsruhe

von

Thomas Englert

Abgabedatum 20.09.2021

Bearbeitungszeitraum	Sechs Monate
Matrikelnummer	2826216
Kurs	TINF19B3
Ausbildungsfirma	Energie Baden-Württemberg AG Karlsruhe
Betreuer der Ausbildungsfirma	Dr. Götz Lichtwald

Erklärung

Ich versichere hiermit, dass ich meine Projektarbeit T2000 mit dem Thema: »Evaluation des praktischen Nutzen von homomorpher Verschlüsselung« selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort Datum

Unterschrift

Abstract

Die homomorphe Verschlüsselung soll es möglich machen, auf verschlüsselten Daten zu operieren. Das Zeil der vorliegenden Arbeit ist, zu evaluieren in wie weit sich die homomorphe Verschlüsselung im Unternehmensumfeld als Anwendung eignet. Dazu werden drei definierte Anwendungsfälle behandelt: Eine Addition mit Ganzzahlen, eine mathematische Berechnung mit Gleitkommazahlen und eine Data-Insights Berechnung. Für die Anwendungsfälle, werden vorhandenen Programmbibliotheken genutzten, deren Anwendung in dieser Arbeit beschrieben ist. Es stellt sich heraus, dass das Rechnen mit verschlüsselten Ganz- und Gleitkommazahlen praktisch anwendbar ist. Bei verschlüsselten Data-Insights Berechnungen eignet sich die homomorphe Verschlüsselung auf Grund von langsamer Laufzeiten noch nicht. Unternehmen, wie Microsoft und Google, die an der homomorphen Verschlüsselung aktiv forschen, stellen in Aussicht, dass Data-Insights Berechnungen ebenfalls in naher Zukunft effizient möglich sein sollen.

Homomorphic encryption should make it possible to operate on encrypted data. The goal of this paper is to evaluate the suitability of homomorphic encryption for enterprise environments. For this purpose, three defined use cases are treated: An addition with integers, a mathematical calculation with floating point numbers and data insights. For the use cases, existing program libraries are used, whose application is described in this work. It turns out that the computation with coded integer and floating point numbers is practically applicable. For encrypted data insights, homomorphic encryption is not yet suitable due to slow runtimes. Companies such as Microsoft and Google, which are actively researching homomorphic encryption, promise that data insights calculations will also be possible efficiently in the near future.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codelistingverzeichnis

Kapitel 1

Einleitung

Diese Projektarbeit wird im Rahmen des dualen Studiums an der Dualen Hochschule Baden-Württemberg Karlsruhe und in Kooperation mit der Energie Baden-Württemberg AG in Karlsruhe geschrieben. Sie ist als Projektberichtes T2000 im Studiengang Informatik / Informationstechnik zu werten. Der Schwerpunkt dieser Arbeit bezieht sich auf die homomorphe Verschlüsselung. Dabei soll in der Arbeit geklärt werden, wie praxistauglich der Einsatz der homomorphen Verschlüsselung bereits sein kann. Betreut wurde die Arbeit durch Herrn Dr. Götz Lichtwald, Leiter des Bereichs Systemkritische Infrastruktur bei der Energie Baden-Württemberg AG.

1.1 Einführung und Motivation

Verschlüsselung kann bei einer schriftlichen Kommunikation genutzt werden, um geheime Informationen nicht direkt lesen zu können. Das hat seinen Ursprung bereits im Kaiserreich von Julius Caeser, mit der Caesar-Chiffre [Tranquillus und Divus Julius Maximilian Ihm o. D.] Mit dem voranschreiten des Zeitalters bzw. dem Entwickeln der Technik (z. B. Funktechnik, Digitaltechnik) sind simple Verschlüsselungsmethoden wie die Caesar-Chiffre in der Digitaltechnik uninteressant. Trotzdem ist es unter gewissen Umständen vonnöten, dass Inhalte von Nachrichten nicht jede Person lesen kann. Daher existiert die Verschlüsselung im Bereich der Kommunikation weiterhin, z. B. Transportverschlüsselung bei E-Mails. Seit dem Zeitalter der Digitalisierung ist die Verschlüsselung ein großer Bestandteil der Kommunikation. Bereits 1978 definiert Ronald L. Rivest ein asymmetrisches Verschlüsselungsverfahren [Rivest, Adleman und Dertouzos 1978], welches heute als Grundlage für verschlüsselte Kommunikation zum Einsatz kommt. Asymmetrische Verschlüsselung ist die Grundlage des Pretty Good Privacy (PGP) [Kuobin 2011] Verfahrens, welches

verwendet wird um E-Mails verschlüsselt zwischen Sender und Empfänger zu versenden. Dabei hat PGP eine wichtige Rolle in der verschlüsselten Kommunikation eingenommen. PGP stellt sicher, dass brisante Informationen von beliebigen Personen nicht in falsche Hände geraten, oder ausgespäht werden.
Der Unterschied den die homomorphe Verschlüsselung, zu den konventionellen und weit verbreiteten Verschlüsselungstechniken bringt ist, dass der Algorithmus hinter der homomorphen Verschlüsselung, Daten (z. B. Zahlen, DNA, Genome, etc.) verändern kann. Dabei bleiben die Daten zu jeder Zeit voll verschlüsselt.
In der Dissertation von [Gentry 2009], vergleicht er die homomorphe Verschlüsselung mit folgendem Beispiel aus der realen Welt:
Wir nehmen an, dass die Inhaberin eines Juweliergeschäfts (Alice) möchte, dass ihre Mitarbeiter Diamanten in ihrem Geschäft schleifen sollen. Dabei hat Alice Angst davor, dass die Mitarbeiter sie bestehlen könnten. Sie löst das Problem, indem sie Handschuhkästchen konstruiert, zu denen nur sie den passenden Schlüssel hat. In diese Handschuhkästchen kann Alice Diamanten hineinlegen. Die Mitarbeiter können durch die Handschuhe die Diamanten schleifen und bearbeiten, aber die Mitarbeiter können die Diamanten nicht entwenden. Den Mitarbeitern fehlt der Schlüssel, um das Kästchen zu öffnen. Die Kästchen sind außerdem durchsichtig, sodass der Mitarbeiter die Diamanten sehen kann. In dieser Analogie bedeutet Verschlüsselung, dass der Mitarbeiter nichts aus der Kiste herausnehmen kann, nicht dass er die Diamanten nicht sehen kann. Wenn der Angestellte fertig ist, kann Alice das fertige Produkt mit Hilfe ihres Schlüssels in aller Ruhe entnehmen.
Zurückübertragen kann sich vorgestellt werden, dass die verschlüsselten Daten die Diamanten sind. Jeder kann sie sehen, was in der verschlüsselten Welt kein Problem darstellt. Durch den Zugang mit den Handschuhen, kann sie auch jeder bearbeiten, der den Zugang hat. Das ist mit dem Evaluation-key bei der homomorphen Verschlüsselung ebenfalls möglich (siehe dazu Kapitel 2.1). Aber nur Alice kann die veränderten Diamanten entnehmen (Analogie zum entschlüsseln), da sie den passenden Schlüssel hat.

Ein weiteres mögliches Anwendungsszenario ist, welches Microsoft auf ihrer Webseite beschreibt, dass Dokumente aus dem Umfeld von Office 365 nicht mehr nur transportverschlüsselt übertragen werden (siehe Bild 1.1). Die Daten, in diesem Beispiel Word-Dokumente, werden verschlüsselt zum Server von Microsoft übertragen und verarbeitet ohne sie zu entschlüsseln (siehe Bild 1.2). Dabei hat nur der Anwender selber die Word-Dokumente unverschlüsselt auf seinem Client vorhanden. Die Vorteile dabei sind, dass Microsoft zu keinem Zeitpunkt die Word-Dokumente entschlüsselt zur Bearbeitung auf

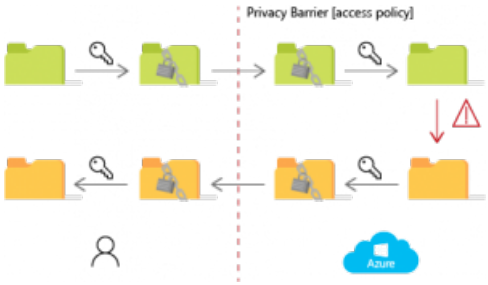

Abbildung 1.1: Daten sind nur bei der Übertragung verschlüsselt. Quelle: [Microsoft 2021]

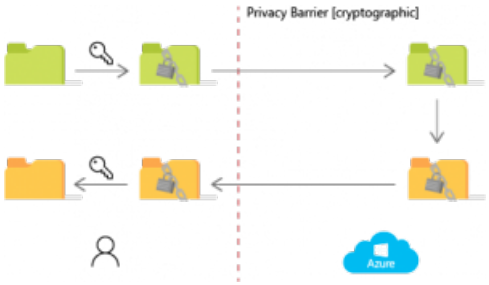

Abbildung 1.2: Daten sind immer verschlüsselt. Quelle: [Microsoft 2021]

ihren Servern liegen hat. Selbst bei einem Datenverlust von Microsoft wären die Daten des Anwenders nicht in Gefahr, da sie verschlüsselt sind. Der Dieb würde nur verschlüsselte Daten erbeuten, die er ohne den Schlüssel des Anwenders nicht entschlüsseln könnte.

Durch die beiden möglichen Szenarien aus den letzten beiden Abschnitten lässt sich die Motivation dieser Arbeit ableiten. In dieser Arbeit ist herauszufinden, ob sich die denkbaren Szenarien aus diesem Abschnitt, schon heute Anwendung finden könnte. Wie eine solche Anwendung aussehen könnte, wird in dieser Projektarbeit in den Kapiteln 2 und 3 erläutert und getestet. Anschließend wird in Kapitel 4 und 5 ein Fazit und Ausblick der Arbeit gegeben.

1.2 Ziel der Arbeit

Durch die im vorherigen Abschnitt erläuterte Motivation der Arbeit lassen sich folgende Ziele definieren:
Es soll einen Gesamtüberblick zum Stand der homomorphen Verschlüsselung verschafft werden, sodass ein Gefühl dafür entwickelt werden kann, wie die Verschlüsselung funktioniert. Dabei soll nur klar werden, wie die Verschlüsselung auf einfacher Ebene funktioniert. Sobald die theoretischen Grundlagen gelegt sind sollen drei konkrete Ziele verfolgt werden:

1. Testen einer Addition mit ganzen Zahlen.
2. Testen einer mathematischen Operation mit Gleitkommazahlen.
3. Evaluieren, ob Data Insights Berechnungen möglich sein können.

Bei dem Testen der drei Szenarien, soll darauf geachtet werden, wie sich die Berechnungen und Methoden anwenden lassen. Wichtig dabei ist, ob es mit dem Stand der Technik von heute schon möglich ist verschlüsselte Operationen auszuführen und wenn ja, wie effizient diese sind.
Das Hauptaugenmerk soll sich dabei auf das Szenario drei beziehen, die Data Insights Berechnung. Diese Methode ist für Unternehmen, welche auf Cloud-Anbieter zurückgreifen besonders interessant. Sie könnten damit Hyperscaler aus der Cloud mit Daten füllen und Berechnungen mit personenbezogenen Daten ausführen lassen. Bzgl. Datenschutz würde das kein Problem darstellen, da die Daten dank der homomorphen Verschlüsselung zu jedem Zeitpunkt verschlüsselt sind. Aus diesem Grund, soll bei der Data Insights Berechnung auf die Anwendbarkeit eingegangen werden.

Kapitel 2

Stand der Technik zur homomorphen Verschlüsselung

In diesem Kapitel werden die Grundlagen zur homomorphen Verschlüsselungen erläutert. Es wird auf den aktuellen Stand der homomorphen Verschlüsselung eingegangen und wie sich die homomorphe Verschlüsselung in den letzten Jahren weiterentwickelt hat. Dabei wird zunächst im Kapitel 2.1 grundlegend erklärt was homomorphe Verschlüsselung ist, bzw. mit welchen Algorithmen sie operieren kann. In Kapitel 2.2 werden zwei konkret umgesetzte und schon vorhandene homomorphe Algorithmen beschrieben. In Kapitel 2.3 wird erklärt wie sicher homomorphe Verschlüsselung einzuschätzen ist. Zum Abschluss werden in Kapitel 2.4 Bibliotheken aufgezählt und verglichen, mit denen homomorphe Operationen ein einer Hochsprache (z. B. C++) ausgeführt werden können. Dabei werden die verschiedenen Bibliotheken evaluiert, welche für diese Arbeit in Betracht gezogen werden soll.

2.1 Was ist homomorphe Verschlüsselung?

Bei der homomorphen Verschlüsselung soll es möglich sein, Operationen mit verschlüsselten Daten und Datensätzen durchzuführen. Dabei können die Operationen simple Rechenalgorithmen sein, Operationen wie Data-Insights Berechnungen. Ein Beispiel ist folgendes: Ein Nutzer lädt seine DNA Daten verschlüsselt auf einen Server hoch. Die Betreiber der Server können die Daten der DNA nicht entschlüsseln und somit nicht lesen. Auf dem Server sind bereits hochgeladene DNA Daten, die ebenfalls verschlüsselt sind, vorhanden. Mittels homomorpher Verschlüsselung kann ein Algorithmus, ausgeführt vom Server, ermitteln welche DNA miteinander Verwand ist. Dieses Beispiel ist bereits von

[*iDASH secure genome analysis competition 2018: blockchain genomic data access logging, homomorphic encryption on GWAS, and DNA segment searching* 2020] umgesetzt.
Es sind drei Hauptarten der homomorphen Verschlüsselung definiert:

- Partially Homomorphic Encryption (PHE): ermöglicht das Anwenden einer mathematischen Operation auf den vorhandenen verschlüsselten Datensatz.
- Somewhat Homomorphic Encryption (SHE): ermöglicht das Anwenden mehrerer mathematischen Operationen auf den vorhandenen verschlüsselten Datensatz.
- Fully Homomorphic Encryption (FHE): ermöglicht es, dass mathematische Operationen beliebig oft auf den vorhanden verschlüsselten Datensatz angewendet werden kann.

In dieser Arbeit wird sich hauptsächlich auf den dritten Typ, die Fully Homomorphic Encryption (FHE) konzentriert und damit gearbeitet.

Homomorphe Verschlüsselung arbeitet nach dem Public-Key-Verschlüsselungsverfahren. Bei diesem Verschlüsselungsverfahren werden zwei Schlüssel, ein Private- (privater) und ein Public-key (öffentlicher) erstellt. Dabei kann eine Nachricht N nur von dem Public-key, verschlüsselt werden. Aus dem Private-key kann jederzeit der Public-key erzeugt werden, so ist es indirekt möglich mit dem Private-key zu verschlüsseln. Die Nachricht kann zu jedem Zeitpunkt nur von dem Private-key entschlüsselt werden. Nehmen wir an es gibt einen Verschlüsselungalgorithmus V und einen Entschlüsselungalgorithmus E dann muss folgendes gegeben sein:

$$E(V(N)) = N$$

Dabei ist der Algorithmus nicht kommutativ. Dieses Verfahren wurde bereits 1978 von Rivers beschrieben und diskutiert. Rivers beschreibt in seinem Paper ebenfalls, wie die Schlüssel erstellt werden können. [Rivest, Adleman und Dertouzos 1978].

Homomorphe Verschlüsselung setzt bei diesem Prinzip an, verwendet jedoch weitere Schlüssel, um wie zu Beginn des Kapitels beschrieben, Operationen auf den verschlüsselten Daten zuzulassen. Um solche Schlüssel zu generieren werden zunächst folgende Parameter in dem Paper [Albrecht u. a. 2018] der Homomorphic Encyption Standard definiert:

- λ bestimmt die Komplexität, bzw. Bitlänge und zugleich die Sicherheit eines durch den Schlüssels erzeugten Ciphertext an. Dabei legt λ die Bitlänge fest, bei einer

128-Bit Sicherheit gilt $\lambda = 128$. Ein 256-Bit langer Ciphertext benötigt mehr Speicher und ist langsamer, als ein 128-Bit langer Chiphertext. Dafür können mehrere Mathematische Operationen auf dem 256-Bit langem Ciphertext ausgeführt werden.

- PT ist der Plaintext Space. In dem Paper [Albrecht u. a. 2018] sind zwei Möglichkeiten zur Erstellung des Plaintext Spaces festgelegt:
  - **Modular Integers (MI)**, bezeichnet Integers innerhalb eines mathematischen Restklassenrings Z_p. Dabei ist p der Modulo Operator. Beispiel: $p = 1024$ bedeutet der Restklassenrings Z modulo 1024 (Z_{1024}) von [0,1023] definiert ist.
  - **Extension Ring/Fields (EX)** sind ähnlich dem MI Restklassenring modulo p, jedoch kommt eine pronominale Funktion $f(x)$ über den Restklassenring Z_p hinzu. Damit ist der Plaintextspace als *Z[x]/(p,f(x))* definiert, gesprochen Z von x modulo p von $f(x)$. Jedes Element innerhalb dieses Restklassenrings ist somit ein Integer Polynom, welches vom Grad kleiner als $f(x)$ sein muss. Die Koeffizienten des Polynoms sind zwischen (0,p-1) definiert. Alle Operationen werden via modulo $f(x)$ und modulo p berechnet.
- K bezeichnet die Dimension der zu verschlüsselten Vektoren. Beispiel: $K = 100$, $PT = (MI, 1024)$, bedeutet, dass die zu verschlüsselnde Nachricht Vektoren $(V_1, ..., V_K)$ sind, wobei V_i aus dem Ring (0,1023) gewählt ist.
- B bezeichnet einen Hilfsparameter, häufig "Noise"genannt, der dazu dient, die Komplexität der Programme zu steuern, die auf verschlüsselten Nachrichten operieren. Niedrigere Parameter stehen für "kleinere", weniger ausdrucksstarke oder weniger komplexe Programme. Niedrigere Parameter bedeuten im Allgemeinen kleinere Parameter für das gesamte Verfahren. Dies führt zu kleineren Chiffretexten und effizienteren Auswertungsverfahren. Höhere Parameter erhöhen im Allgemeinen die Schlüsselgröße, die Größe des Chiffretextes und die Komplexität der Auswerteverfahren. Höhere Parameter sind notwendig, um komplexere Programme auszuwerten.

a. Public-key Erzeugung

Mit Hilfe dieser festgelegten Parametern (λ, PT, K, B), wird ein Secret-key (SK), Private-key (PK) und ein Evaluation-key (EK) erstellt [Albrecht u. a. 2018]. Der Algorithmus muss wie folgt definiert sein und nennt sich **Public-key Erzeugung**:

$$\text{PubKeygen}(\lambda,\ PT,\ K,\ B) \rightarrow \text{SK, PK, EK}$$

Der PK kann benutzt werden um Nachrichten zu verschlüsseln. Mit dem PK können Nachrichten nicht entschlüsselt werden, wie zu Beginn des Kapitels beschreiben.
Im Vergleich zu bisher existierenden Public-Private-Verschlüsselungsverfahren, ist der EK der neue zusätzliche Schlüssel. Der EK wird verwendet um homomorphe Operationen auf Nachrichten oder auch Daten durchzuführen. Daten können nicht mit dem PK oder dem EK entschlüsselt werden, auch nicht wenn einem Algorithmus zur Entschlüsselung beide (PK und EK) bekannt sind.

b. Secret-key Erzeugung
Der Algorithmus erzeugt den Secret-key. Dieser Secret-key wird benötigt, um Nachrichten durch das Schema sowohl zu verschlüsseln als auch zu entschlüsseln.

$$\text{SecKeygen}(\lambda,\ PT,\ K,\ B) \rightarrow \text{SK, EK}$$

Er muss vom Benutzer geheim gehalten werden. Der Algorithmus erzeugt zusätzlich einen Evaluation-key, der benötigt wird, um homomorphe Operationen über die Ciphertexte durchzuführen. Der Evaluation-key kann jedem gegeben werden, der homomorphe Operationen auf den Ciphertexten durchführten will. Eine Person, die nur den Evaluation-key hat, kann die eigentliche unverschlüsselte Nachricht aus den Ciphertexten nicht entschlüsseln.

c. Public Verschlüsselungs Algorithmus
Ein weiterer Algorithmus ist der **Public Verschlüsselungs Algorithmus**. Er nimmt als Eingabe den PK und eine Nachricht N und liefert einen Ciphertext C [Albrecht u. a. 2018].

$$\text{PubEncrypt}(PK,\ N) \rightarrow C$$

d. Secret Key Verschlüsselung
Der Algorithmus **Secret Key Verschlüsselung** arbeitet ähnlich wie der Public Verschlüsselungs Algorithmus [Albrecht u. a. 2018].

$$\text{SecEncrypt}(SK,\ N) \rightarrow C$$

e. Entschlüsselungs Algorithmus
Der **Entschlüsselungs Algorithmus** ist wie folgt definiert.

$$\text{Decrypt}(SK,\ C) \rightarrow N$$

f. Additions Algorithmus

Eine Addition zweier Ciphertexte ist mithilfe des EK möglich. Der **Additions Algorithmus** muss wie folgt definiert sein. Dabei sind Params die zu Beginn des Kapitels beschriebenen wurden nötig (λ, PT, K, B).

$$\text{Add}(\text{Params},EK,C1,C2) \rightarrow C3$$

Die gleiche Operation ist auch mit einer Nachricht N anstelle des Ciphertextes $C2$ möglich. Für weitere Informationen siehe in dem Paper der Homomorphic Encryption Organisation [Albrecht u. a. 2018].

g. Multiplikations Algorithmus

Ein **Multiplikations Algortihmus** ist wie der Additions Algorithmus definiert.

$$\text{Mul}(\text{Params},EK,C1,C2) \rightarrow C3$$

h. Refresh Algorithmus

Ein zusätzlicher Algorithmus der gegeben werden muss, ist der **Refresh Algorithmus**. Die Aufgabe des Refresh Algorithmus ist, dass zu groß (im Bezug auf die Bitlänge) gewordene Ciphertexte vereinfacht werden, sodass der Plaintext Space kleiner wird und mit der Verschlüsselung performanter gearbeitet werden kann. Der Algorithmus bekommt als zusätzlichen Parameter ein Flag übergeben, welches entscheidet, wie der Ciphertext erneuert wird, siehe dazu [Albrecht u. a. 2018].

$$\text{Refresh}(\text{Params, flag}EK,\ C1) \rightarrow C2$$

In dem Paper der Homomorphic Encryption Organisation [Albrecht u. a. 2018] sind weitere Algorithmen diskutiert. Sie werden hier aber nicht weiter beschrieben, da sie nicht von Bedeutung für den praktischen Teil (siehe Kapitel 3.3) dieser Arbeit sind.

2.2 homomorphe Algorithmen BFV und CKKS

In diesem Abschnitt wird auf zwei konkrete Verfahren eingegangen, mit denen homomorphe Verschlüsselung genutzt werden kann.

Brakerski/Fan-Vercauteren (BFV)

BFV eignet sich gut um Integerberechnungen auszuführen [*Microsoft SEAL (release 3.6)*

2020].
Bei BFV bleiben die Funktionen und Params wie im Kapitel 2.1 definiert. Das bedeutet, dass die Params weiterhin aus λ, PT, K, B bestehen. Zusätzlich kommen folgende Params hinzu[Albrecht u. a. 2018]:

- Key und ErrorDistribution D_1, D_2.
- Integer T und $L = \log_T q$
- Integer $W = [q/p]$

BFV verwendet die definierten Algorithmen aus dem Kapitel vorher: PubKeygen, PubEncrypt usw. In dem Paper der Homomorphic Encryption Organistation [Albrecht u. a. 2018] ist beschreiben, wie die einzelnen Schlüssel und Operationen erstellt bzw. ausgeführt werden.

The Cheon-Kim-Kim-Song (CKKS)

Im Gegensatz zu BFV ist es mit CKKS möglich reelle Zahlen zu berechnen. CKKS eignet sich besonders für maschinelles Lernen auf Basis von verschlüsselten Daten, aber auch für das Berechnen von Entfernungen zweier verschlüsselter Standorte [*Microsoft SEAL (release 3.6)* 2020]. Dabei kann die Entfernung beispielsweise die Anzahl von Knoten sein, die passiert werden müssen.
CKKS macht diese Berechnungen mit Hilfe von Chain-Switching möglich. Beim Chain-

```
Line 285 --> Compute PI*x^3 + 0.4*x + 1.
Line 295 --> Decrypt and decode PI*x^3 + 0.4x + 1.
    + Expected result:

    [ 4.5415927, 26.9327412, 87.0230016 ]

    + Computed result ..... Correct.

    [ 4.5415957, 26.9327649, 87.0230813 ]
```

Abbildung 2.1: Berechnungsfehler mittels CKKS Algorithmus

Switching werden von vorher erzeugen Hashes der Primzahlen, Primzahlen aus der Chain entfernt und neue Hashes berechnet. Der Vorteil davon ist, dass Berechnungen viel effizienter und schneller ausgeführt werden können (siehe dazu [*Microsoft SEAL (release 3.6)* 2020] Datei 3_levels.cpp und Kapitel 3.2).

Ein Problem von CKKS ist, das die Berechnungen auf Grund von Fehlern nicht genau sind. Siehe 2.1, hier wurde die Funktion $\pi \cdot x^3 + 0,4 \cdot x + 1$ auf Zahlen kleiner 3 angewendet. Gut zu erkennen ist, dass umso größer die Zahl wird, umso größer ist die Abweichung. Für weitere Informationen über das Verhalten mit Berechnungen von CKKS empfiehlt sich die Literatur von [Cheon u. a. 2016] bzw. [*Microsoft SEAL (release 3.6)* 2020].

2.3 Sicherheit von homomorpher Verschlüsselung

Wie bereits in Kapitel 2.1 beschrieben, basiert die homomorphe Verschlüsselung auf dem Public-Private-Key Prinzip. Dieses Prinzip wird auch asymmetrisches Verschlüsselungsverfahren genannt. Wenn der Hinweg einer Primzahlmultiplikation einfach ist, der Rückweg, also die Faktorisierung aber aufwendig ist, spricht man von Falltürfunktionen. Asymmetrische Verfahren beruhen auf Falltürfunktionen. Dabei ist die Funktion in eine Richtung leicht berechenbar (z. B. für die Schlüsselgenerierung) aber in die andere Richtung nicht (kein Rückschluss auf die Schlüssel) [Drehling 2021]. Zusätzlich kommt hinzu, dass asymmetrische Verschlüsselungsverfahren auf dem Problem der Komplexitätstheorie $P \neq NP$ basieren. Solange $P \neq NP$ gilt, sind asymmetrische Verschlüsselungsverfahren auch sicher. Aus den genannten Gründen gilt die homomorphe Verschlüsselung, zum jetzigen Zeitpunkt (Jahr 2021) als sicher [Rivest, Adleman und Dertouzos 1978].
Ein Problem ist, das in Zukunft auf die homomorphe Verschlüsselung und deren Sicherheit zukommen wird, dass sie gegen die Rechenleistung von Quantencomputern sicher sein muss. Damit die homomorphe Verschlüsselung sicher gegen Quantencomputern ist wird das Ring Learning With Errors (RLWE) Problem angewendet. Das RLEW basiert auf dem allgemeinem Learning With Errors (LWE) Problem, wobei noch eine zusätzliche Matrix mit spezieller algebraischer Struktur hinzugefügt wird [Chase u. a. 2017].
Bei dem LWE-Problem handelt es sich um ein mathematisches Problem, welches dem lösen von Gleichungssystemen sehr nahe kommt. Dabei wird jedoch ein zusätzlicher Vektor in das System mit eingebaut, so dass eine Lösung des Gleichungssystems mit dem Gaußschen Algorithmus nicht mehr möglich ist, beispielsweise:

$$4x + 5y - 20z = 50 (\mathrm{mod} 220)$$

Die Lösung der Gleichung mittels des LWE-Problem führt jedoch zu keinem korrektem Ergebnis. Der Ersteller des Private-key hat kleine Fehler (+2,-1+2,...) in die Gleichung eingefügt, sodass das Ergebnis der obigen Gleichung niemals 50 ergeben kann. Die wahren Lösungen sind nur mittels des Private-key zu erhalten. Das reicht zum Verschlüsseln, zum

Entschlüsseln sind jedoch die richtigen Werte von nöten [Drehling und Tremmel 2021]. Das LWE-Problem wurde jedoch erst 2005 von Oded Regev das erste mal beschreiben [Regev 2005]. Daher wird aktuell nur angenommen, dass es sicher ist, weil die Verschlüsselung bisher noch nicht geknackt wurde, einen mathematischen Beweis dazu gibt es nicht [Regev 2005]. Im Security Standard der homomorphic encryption Organisation sind weitere Rahmenbedingungen zum LWE und RLWE diskutiert.
Zusätzlich gilt, dass Quantencomputer Public-Key-Verschlüsselungsverfahren nur um eine Komplexitätsklasse schneller berechnen können. Der Quantencomputer kann dadurch eine asymmetrische Verschlüsselung nur mit quadratischem Zeitaufwand schneller berechnen [Grävemeyer 2021]. Aus den genannten Gründen, könnte einen RSA Schlüssel von 256 Bit Komplexität, auf 512 Bit verlängern. Somit wäre die Verschlüsselung für Quantencomputer genauso schwer zu berechnen, wie für nicht Quantencomputer.

2.4 Bibliotheken zur praktischen Anwendung von homomorpher Verschlüsselung

Auf der Webseite der Homomorphen Encryption Organisation sind Bibliotheken verlinkt, mit denen bereits homomorphe Verschlüsselungsalgorithmen aus Kapitel 2.2 in hohen Programmiersprachen (z.B. C++, Go) umgesetzt sind. Einige der Bibliotheken sind nicht mehr gepflegt, aus diesem Grund wurden nur aktuelle heran gezogen. Als aktuell gilt jede Bibliothek, die in den letzten zwölf Monaten einen Commit bekommen haben. Um zu Bewerten mit welcher Bibliothek das Ziel der Projektarbeit am Besten erreicht werden kann, wurde eine Tabelle mit Features aufgestellt, welche bewertet ob die Bibliothek einen Algorithmus / Feature implementiert hat, oder nicht (siehe 2.1).

Kriterium	Microsoft SEAL	Lattigo	PALISADE	Google (TFHE)
BFV	X	X	X	unbekannt
CKKS	X	X	X	unbekannt
Stringvergleiche	-	-	-	X

Tabelle 2.1: Auswertung von verwenderter Bibliothek für homomorphe Verschlüsselung

Bei der Bewertung (Tabelle 2.1) der Kriterien sind die Punkte wie folgt eingeflossen:

- BFV - ist der Algorithmus in der Bibliothek implementiert oder nicht.
- CKKS - ist der Algorithmus in der Bibliothek implementiert oder nicht.

- Stringvergleiche - ist es mit der Bibliothek möglich Strings miteinander zu vergleichen oder nicht.

Microsoft SEAL
Microsoft SEAL (Simple Encrypted Arithmetic Library) ist eine OpenSource Bibliothek für verschiedene Algorithmen der homomorphen Verschlüsselung. Der Source Code ist auf Github veröffentlicht. Dabei ist die Bibliothek unter allen bekannten Betriebssystem anwendbar (Linux, MacOS, Windows). Die Bibliothek hat die Algorithmen BFV und CKKS implementiert und verwendet dabei die Programmiersprachen C++, C#, Python, JavaScript und TypeScript [*Microsoft SEAL (release 3.6)* 2020].

Lattigo
Lattigo ist ein in Go geschriebenes Modul, das homomorphe Verschlüsselung auf Basis des RLWE Problems implementiert hat. Die Bibliothek bietet die Algorithmen BFV und CKKS an [*Lattigo v2.2.0* 2021].

PALISADE
PALISADE ist eine OpenSource Bibliothek, die eine effiziente Implementierungen von homomorphen Verschlüsselungsverfahren bietet. PALISDAE bietet folgende implementierte Algorithmen an: BFV, BGV, CKKS, FHEW, TFHE. Die Bibliothek ist in den Programmiersprachen C++, JavaScript und Python geschrieben [Li und Micciancio 2020].

Google (TFHE)
Die Bibliothek von Google hat den Vorteil, dass es einen Transpiler für C++ Code gibt. Es ist mit dem Transpiler von Google dadurch möglich, Code zu schreiben, der durch den Transpiler in die TFHE Bibliothek von Google übersetzt wird. Google gibt dabei nicht an, welche Algorithmen der TFHE Bibliothek zugrunde liegen [*Fully Homomorphic Encryption (FHE)* 2021].

Die Auswertung zeigt, dass im Kapitel 3 die Beispiele mittels Microsoft SEAL, Lattigo oder PALISADE umgesetzt werden könnten. Alle drei Bibliotheken ermöglichen das Rechnen mit dem BFV und CKKS Algorithmus. In dieser Arbeit die Tests in Kapitel 3 mit der Microsoft SEAL Bibliothek ausgeführt.

Dabei ist anzumerken, dass die Microsoft SEAL Bibliothek vor allem auf Integeroperationen mit dem BFV und CKKS Scheme ausgelegt ist (siehe Kapitel 2.2).

Das in Kapitel 1.2 bestimmte Ziel soll Stringoperationen ausführen können. In der Tabelle 2.1 ist zu entnehmen, dass sich dazu nur die Bibliothek von Google eignet. Aus diesem Grund wird für Stringmanipulationen in erster Linie die Bibliothek von Google verwendet. Google hat im Juni 2021 einen Transpiler für die Bibliothek TFHE: Fast Fully Homomorphic Encryption over the Torus herausgegeben [Guevara 2021].

Der Transpiler von Google funktioniert in fünf Schritten [*Fully Homomorphic Encryption (FHE)* 2021]:

1. Die XLS[cc]-Phase übersetzt eine Funktion in C++ geschrieben in eine XLS intermediate representation (IR).

2. Die Optimierungsphase Optimiert XLS IR Code.

3. Die Booleanifier-Phase übersetzt den XLS IR Code in boolesche XLS IR Repräsentation (z. B. AND, OR, NOT).

4. Die FHE IR Phase übersetzt die boolesche XLS IR Repräsentation in FHE-C++ Code aus der Bibliothek von Google.

5. Die TFHE Testbench-Phase führt den FHE-C++ Code aus. Dabei ist eine Testbench-Datei vom Nutzer des Transpilers immer selbst zu erstellen.

Da die Arbeit sich auf die Anwendbarkeit von homomorpher Verschlüsselung beschränkt, wir der Transpiler nicht weiter erläutert. Die Funktionsweise von XLS IR kann in folgender Dokumentation gelesen werden: [Google o. D.] Der Transpiler ist im Github direkt genauer erläutert: [*Fully Homomorphic Encryption (FHE)* 2021].

2.5 Zusammenfassung der Technischen Grundlagen

In diesem Kapitel, werden die Erkenntnisse des Kapitel 2 kurz zusammengefasst.
Die homomorphe Verschlüsselung ist eine Public-Privat-key Verschlüsselung, welche mit Funktionen erweitert wurde, um Rechenoperationen auf den verschlüsselten Daten auszuführen (siehe Kapitel 2.1). Dabei ist vor allem wichtig, dass ein weiterer Schlüssel zu dem Public-Privat-key Verfahren hinzugekommen ist - der Evaluation-key. Der Schlüssel wird von dem Nutzer benötigt, der eine verschlüsselte mathematische Addition ausführen möchte. Dabei kann mit dem Schlüssel wie mit einem Public-key umgegangen werden. Sprich nur mit dem Evaluation-key lassen sich verschlüsselte Daten nicht entschlüsseln.
Um verschlüsselte Berechnungen mit Ganz- und Gleitkommazahlen auszuführen, wird dabei auf die beiden aktuell am meisten genutzten Algorithmen BFV und CKKS zurückgegriffen (siehe Kapitel 2.2).
Die Sicherheit der homomorphen Verschlüsselung basiert dabei auf der Sicherheit der asymmetrischen Verschlüsselung. Solange die als sicher gilt, ist es die homomorphe Verschlüsselung auch (siehe Kapitel 2.3). Um Quantencomputern sicher zu verschlüsseln, kann ggf. das RLWE-Problem verwendet werden, oder die Komplexitätsklasse um eins erhöht werden.
Um den praktischen Teil in Kapitel 3 durchzuführen, wurde in Kapitel 2.4 festgelegt, dass sich dazu am besten die Bibliotheken von Microsoft und Google eigenen. Die Bibliotheken bieten einen großen Funktionsumfang und sind bereits heute einfach einzusetzen.

Kapitel 3

Praktischer Nutzen der homomorphen Verschlüsselung

In diesem Kapitel soll die praktische Nutzung von homomorpher Verschlüsselung evaluiert werden. Dabei werden drei Anwendungsfälle definieren und näher beschrieben:

- Addition mit zwei verschlüsselten natürlichen Zahlen
- Multiplikation mit Gleitkommazahlen
- Berechnung einer Häufigkeit eines festgelegten Parameter in einem Datensatz

Bei der Addition sollen zwei Integerzahlen mit Hilfe des in Kapitel 2.2 beschriebenen BFV Schemas addiert werden. Das BFV Schema kann nur mit Ganzzahlen genutzt werden. Es liefert dafür immer richtige Ergebnisse.
Die Addition mit Gleitkommazahlen werden mit dem in Kapitel 2.2 beschriebenen CKKS Schema durchgeführt. Der Nachteil bei CKKS ist, dass die errechneten Werte kleine Abweichungen haben können, gerade bei Gleitkommazahlen. Zur Zeit ist CKKS jedoch das einzige Schema mit dem überhaupt Gleitkommaberechnungen homomorph Verschlüsselt durchgeführt werden kann (siehe Kapitel 2.2). Deswegen kommt es in diesem Versuch zum Einsatz.
Bei dem letzten Versuch in Kapitel 3.3 wird die Frage geklärt, in wie weit sich homomorphe Verschlüsselung schon auf ein konkretes Problem, was in der Praxis einen Nutzen hat, anwenden lässt. Dazu wird ein Datensatz von 30 Kunden mit Namen und Postleitzahl erstellt. Eine übliche Methode zur Verwaltung der Daten soll mit dem Transpiler von Google (siehe Kapitel 2.4) vergleichen werden.
In jedem der Kapitel wird bei der Berechnung die Zeit gestoppt, um am Ende in Kapitel

3.4 ein Fazit aller Tests zu ziehen. Dabei wird für alle Tests folgende Hard- und Software verwendet:

- Betriebssystem: Arch Linux
- CPU: AMD Ryzen 5 3500U
- RAM: 14,6 GB
- Festplattentyp: NVMe SSD

Zusätzlich wird versucht alle Tests mindestens 1.000 Mal auszuführen und anschließend der Mittelwert der Laufzeit berechnet. Im Anschluss wird der Durchlauf der Tests mindestens verdoppelt, sodass ein Gefühl für die Laufzeit der Algorithmen und des Verschlüsselungsschema entstehen kann.

3.1 Addition zweier verschlüsselter ganzen Zahlen

In diesem Kapitel wollen wir evaluieren, ob und wie es möglich ist zwei ganze Zahlen zu addieren, welche verschlüsselt sind. Dazu greifen wir auf die Bibliothek Microsoft SEAL zurück, wie im Kapitel 2.4 beschrieben. Mit der Addition soll gezeigt werden, dass es überhaupt möglich ist homomorph verschlüsselte Berechnungen durchzuführen. Sobald gezeigt ist, dass eine Addition möglich ist, sind in der Regel auch weitere Operationen möglich. Im Allgemeinen steht danach fest, dass es möglich ist mathematische Operationen mit der homomorphen Verschlüsselung durchzuführen.
In Kapitel 2.1 ist beschreiben, wie ein Additions-Algorithmus auszusehen hat. Mit Hilfe von Microsoft SEAL erstellen wir die Params, aus denen wir wiederum den SK,PK und EK erstellen. Die einfache Addition soll mit dem BFV Algorithmus (siehe Kapitel 2.2), der für ganze Integerzahlen gut geeignet ist, durchgeführt werden. Das Codebeispiel A.1 zeigt wie das Konzept von Kapitel 2.1 umgesetzt werden kann. In den Zeilen 1-7 von Codelisting 3.1 werden die Parameter gesetzt. Zeile 2 und 3 von Codelisting 3.1 geben den Grad des polynominalen Modulo an. Ein höherer Wert von *poly_modulus_degree* vergrößert den Ciphertext Speicherplatz. Alle Operationen werden dadurch langsamer, ermöglicht jedoch kompliziertere verschlüsselte Berechnungen. Empfohlene Werte für *poly_modulus_degree* sind 1024, 2048, 4096, 8192, 16384, 32768. Es ist auch möglich diese Werte zu überschreiten, dabei muss beachtet werden, dass wie bereits erwähnt die Zeit zur Berechnung von Daten und der Speicherbedarf stark ansteigen.

```
1 EncryptionParameters parms(scheme_type::bfv);
2 size_t poly_modulus_degree = 4096;
3 parms.set_poly_modulus_degree(poly_modulus_degree);
4 parms.set_coeff_modulus(CoeffModulus::BFVDefault(poly_modulus_degree));
5 parms.set_plain_modulus(1024);
6
7 SEALContext context(parms);
```

Listing 3.1: Parameter festlegen in Microsoft SEAL mit BFV

Zeile 4 von Codelisting 3.1 setzt den Wert für die Dimension der Primzahl Vektoren (vergleiche Kapitel 2.1 Wert K). Sie sind eine große Ganzzahl, die aus unterschiedlichen Primzahlen mit einer Größe von jeweils bis zu 60 Bit besteht. Der Wert K wird als Vektor dieser Primzahlen dargestellt. Anhand diesen Wertes wird der komplette Ring für die Verschlüsselung vom Microsoft SEAL erzeugt. Die Bitlänge von *coeff_modulus* ist die Summe aller Bitlängen der Primfaktoren. Die folgende Tabelle zeigt das Verhältnis der Werte *poly_modulus_degree* und den maximalen *coeff_modulus* Bit-Längen [*Microsoft SEAL (release 3.6)* 2020]:

poly_modulus_degree	coeff_modulus
1024	27
2048	54
4096	109
8192	218
16384	438
32768	881

Tabelle 3.1: Verhältnis der Parameter poly_modulus_degree und coeff_modulus

Als Beispiel: *poly_modulus_degree* hat den Wert 4096, dann können in *coeff_modulus* maximal drei Primzahlen von jeweils 36 Bit enthalten sein ($3 * 36 = 108$ Bits). Microsoft SEAL verfügt über eine Hilfsfunktion zur Auswahl des *coeff_modulus* (*CoeffModulus::BFVDefault(poly_modulus_degree)*, siehe Codelisting A.1 Zeile 5).

In Zeile 5 von von Codelisting 3.1 wird die Größe des Plaintext Modulo festgelegt. Er bestimmt wir groß der Speicher eines Plaintextes sein darf und legt den Noise Parameter fest (vergleiche Kapitel 2.1 Parameter B)).

Die Parameter werden mit Hilfe des *SEALContext* (Zeile 7 von Codelisting 3.1) evaluiert, sodass keine falsche Eingabe entstehen kann.

Da in dem Beispiel auf den Algorithmus BFV zurückgegriffen wird, wird kein EK benötigt,

```
1 KeyGenerator keygen(context);
2 SecretKey secret_key = keygen.secret_key();
3 PublicKey public_key;
4 keygen.create_public_key(public_key);
5
6 Encryptor encryptor(context, public_key);
7 Evaluator evaluator(context);
8 Decryptor decryptor(context, secret_key);
```

Listing 3.2: Erzeugen von PK,SK,Encryptor, Evaluator und Decryptor in Microsoft SEAL mit BFV

siehe Kapitel 2.2. Die Bibliothek Microsoft SEAL stellt Methoden in der Bibliothek zur Verfügung, mit denen das Erstellen von PK und SK funktionieren (siehe Codelisting 3.2 Zeile 1-4).
Der Encryptor ist für die Entschlüsselung von Ciphertexten verantwortlich. Der Evaluator führt mathematische Operationen auf Ciphertexten, oder Ciphertexten mit Plaintexten aus. Der Decryptor benötigt den SK um eine Entschlüsselung von Daten durchzuführen. Die drei Werkzeuge sind im Codelisting 3.2 in den Zeilen 6-8 zu finden.
Die beiden zu addierenden hexadezimalen Zahlen werden zunächst als Integer festgelegt

```
1 int x = 8;
2 int y = 19;
3 Plaintext x_plain(to_string(x));
4 Plaintext y_plain(to_string(y));
```

Listing 3.3: Zu addierende Integer in hexadezimal

und in einen Plaintext umgewandelt Codelisting 3.3 Zeile 1-4. Der Speicherbedarf für den Ciphertext wird reserviert und die beiden Zahlen x und y entsprechend an diese Speicherstellen verschlüsselt abgespeichert Codelisting 3.4 Zeile 1-5. Mit der Methode *add* vom Evaluator können die beiden verschlüsselten Zahlen miteinander addiert werden (siehe Codelisting 3.4 Zeile 8). Das könnte beispielsweise ebenfalls mit einer Plaintextzahl vom Evaluator gemacht werden, dazu wird die Funktion *add_plain* des Evaluators verwendet. Bei der Funktion *add* wird das Ergebnis der Addition in einem dritten Ciphertext gespeichert. Deswegen muss der in Codelisting 3.4 Zeile 3 zusätzlich angelegt werden. Wird die Funktion *add_inplace* des Evaluators verwendet kann das Ergebnis direkt im zuerst angegebenen Ciphertext gespeichert werden.
Bei der Entschlüsselung der Addition wird der Decryptor das Ergebnis entschlüsseln und

```
1 Ciphertext x_encrypted;
2 Ciphertext y_encrypted;
3 Ciphertext x_plus_y_encrypted;
4 encryptor.encrypt(x_plain, x_encrypted);
5 encryptor.encrypt(y_plain, y_encrypted);
6
7 evaluator.add(x_encrypted, y_encrypted, x_plus_y_encrypted);
```

Listing 3.4: Verschlüsselte Integerberechnung mit der Methode add

in einem Plaintext speichern. Dazu wird der Plaintext angelegt (Codelisting 3.5 Zeile 1) und mit der Funktion *decrypt* (Codelisting 3.5 Zeile 3) das Ergebnis aus Codelisting 3.4 Zeile 7 gespeichert. Mittels der Standardfunktion *cout* von C++ wird das Ergebnis

```
1 Plaintext decrypted_result;
2 decryptor.decrypt(x_plus_y_encrypted, decrypted_result);
3 cout << "0x" << decrypted_result.to_string() << endl;
```

Listing 3.5: Entschlüsselung einer Zahl

in Codelisting 3.4 Zeile 3 ausgegeben. Zusätzlich wird noch *0x* auf die Konsole ausgegeben, damit dem Anwender klar ist, dass das Ergebnis wiederum eine hexadezimale Zahl ist.

Um ein Gefühl für die Performance für den Algorithmus und der homomorphen Verschlüsselung zu bekommen, wird die Addition aus Codelisting 3.4 in einer for-Schleife 1.000 mal ausgeführt (siehe Codelisting 3.6). Dabei wird vor und nach der Ausführung die Zeit gestoppt und das Ergebnis aufgeschrieben. Dieser Vorgang wird fünf mal durchgeführt und anschließend die Berechnungen in der for-Schleife auf 10.000 erhöht. Die Zeiten in Tabelle 3.6 sind so schnell, dass behauptet werden kann, der Algorithmus ist performant und könnte in der Praxis Anwendung finden. BFV eignet sich sehr gut für die Berechnung von ganzen Integerzahlen und das Ergebnis der Berechnung ist korrekt. Anzumerken ist, dass bei der Erhöhung der Berechnungen in der for-Schleife um den Faktor zehn, die Laufzeit ebenfalls ca. verzehnfacht (siehe Tabelle 3.2).

Durchlauf	Zeit (1.000 Berechnungen)	Zeit (10.000 Berechnungen)
1	1.81207 sec	14.3858 sec
2	2.06981 sec	13.8399 sec
3	1.48275 sec	13.9273 sec
4	1.38182 sec	13.985 sec
5	1.39322 sec	13.7421 sec
Mittelwert	**1.62793 sec**	**13.9759 sec**

Tabelle 3.2: Berechnungsdauer homomorph verschlüsselte Ganzzahlen mit dem BFV Schema

```
double cpu_start_time = clock();
for(int i = 0; i < 1000; i++){
        cout << "   number i: " << i << endl;
        Plaintext i_plain(to_string(i));
        Ciphertext i_encrypted;
        encryptor.encrypt(i_plain, i_encrypted);
        if (i == 0){
                evaluator.add(
                x_encrypted, i_encrypted, x_plus_i_encrypted
                );
        }else{
                evaluator.add(
                x_plus_i_encrypted, i_encrypted, x_plus_i_encrypted
                );
        }
}
double cpu_end_time = clock();
```

Listing 3.6: Messen der Brechungsdauer einer Addition mit BFV bei 1.000 Wiederholungen

3.2 Addition zweier verschlüsselter Gleitkommazahlen

In diesem Kapitel soll mit komplexen Gleitkommazahlen gerechnet werden. Als Schema soll CKKS getestet werden, da es sich wie in Kapitel 2.2 beschrieben, am besten dazu eignet Gleitkommazahlen zu verrechnen.
CKKS wird mit Hilfe von Leveling in Microsoft SEAL implementiert. Jedes Mal wenn die Parameter aus Kapitel 2.1 erzeugt werden, werden diese in Microsoft SEAL gehasht und mit einer ID versehen. Zusätzlich erzeugt Microsoft SEAL eine Kette (Chain) von den gehashten Parametern, die sogenannte *modulus switching chain*. Bei jedem neuem Kettenmitglied wird der Modulo Faktor des Polynoms um eins reduziert (siehe Kapitel 2.1 Plaintext Space, Extension Rings/Fields) und die letzte Primzahl der Kette entfernt (genannt special prime). Jedes neue Level der Kette wird mit einem Index versehen und die dort liegenden Parametern sind zu jedem Zeitpunkt abrufbar. Gehen wir davon aus, das unser Schema vier Level hat. Dann beinhaltet das höchste Level alle Schlüssel (PK, EK, SK) und wird *key level* genannt. Das darunter liegende Level 3, ist das *data level*. In der Regel liegen in diesem Level die Daten. Es können sich aber auch Schlüssel und Daten im oberstem Level befinden. Das letzte Level 0 ist das *lowest level* (siehe Abbildung 3.1). Der negative Seiteneffekt davon ist, dass beim Entfernen der letzten Primzahl der Speicherplatz des Ciphertextes reduziert wird. Im Regelfall möchte der Entwickler seiner

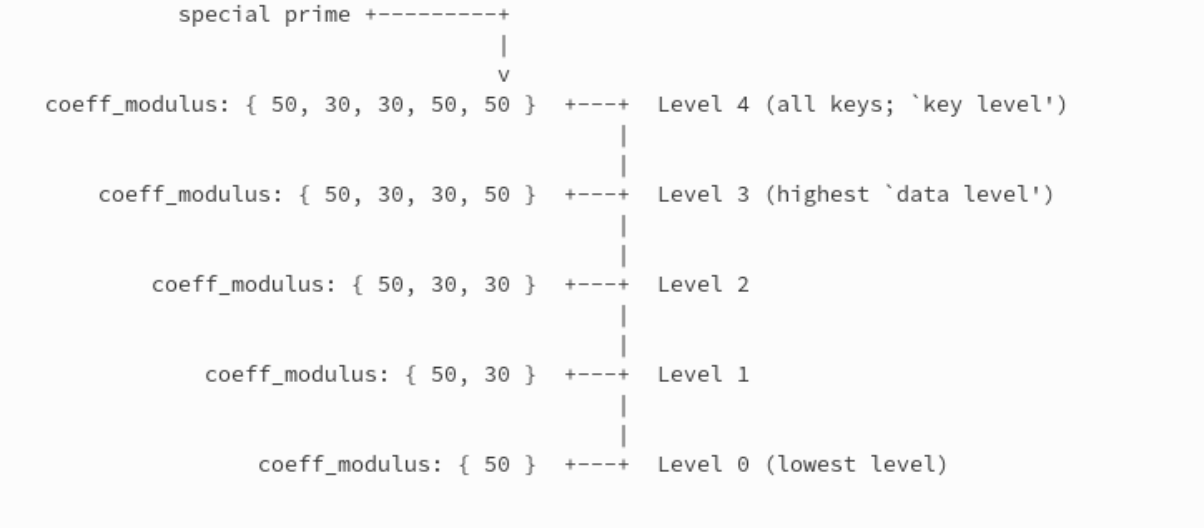

Abbildung 3.1: Leveling in BFV und CKKS. Quelle: [*Microsoft SEAL (release 3.6)* 2020]

Anwendung über Speicherbedarf des Ciphertextes selbst verwalten, deswegen müssen die

coeff_modulus bei CKKS mit Bedacht gewählt werden. Eine Hilfsfunktion wie bei Kapitel 3.1, die optimale Primzahlen auswählt, gibt es nicht.
Nehmen wir an, S sei die Skala für den Speicherbedarf des Chiphertexts und P die letzte Primzahl. Beim Skalieren auf das nächst untere Level wird die Skala auf S/P bestimmt und die Primzahl P entfernt (siehe Abbildung 3.1 coeff_modulus). Die Anzahl der Primzahlen begrenzt dabei wie oft der Prozess durchgeführt werden kann.
Prinzipiell ist es möglich den Wert S frei zu wählen. Jedoch empfiehlt es sich P und S sehr nahe beieinander zu wählen. Denn Ciphertexte sind vor einer Multiplikation S groß, nach einer Multiplikation S^2 und wenn der Ciphertext auf das untere Level geschoben wurde S^2/P. Wenn nun S und P sehr nahe beieinander liegen, hat das zu Folge, dass S wieder sehr nahe seinem ursprünglichem Wert ist.
Um gute *coeff_modulus* zu bestimmt, empfiehlt Microsoft SEAL folgende drei Schritte [*Microsoft SEAL (release 3.6)* 2020]:

1. Die erste Primzahl der *coeff_modulus* soll 60-Bit groß sein. Das ergibt ein gute Genauigkeit beim Entschlüsseln des Ciphertextes.

2. Ist die erste Primzahl 60-Bit groß, soll die letzte Primzahl ebenfalls 60-Bit groß sein. Der Grund dafür ist, dass die letzte Primzahl wie bereits beschreiben die special prime ist und deswegen immer mindestens genauso groß wie die größte Primzahl sein sollte.

3. Die Primzahlen sollen möglichst nahe beieinander liegen.

In dem Codelisting A.2 sollen die bisherigen Grundlagen genauer erläutert werden. Berechnet wird $\pi * x^3$. Die Zeilen 1-6 von Codelisting 3.7 ähneln sich dem Codelisting von

```
1 EncryptionParameters parms(scheme_type::ckks);
2 size_t poly_modulus_degree = 8192;
3 parms.set_poly_modulus_degree(poly_modulus_degree);
4 parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 60, 40, 40, 60 }))
5 double s = pow(2.0, 40);
6 SEALContext context(parms);
```

Listing 3.7: Parameter festlegen in Microsoft SEAL auf Basis von CKKS

3.1, jedoch unterschieden sich folgende Zeilen: In Zeile 4 werden die Primzahlen wie zuvor in diesem Kapitel beschreiben selber gesetzt, da es keine Hilfsfunktion, wie bei BFV gibt. Zusätzlich wird in Zeile 5 die Skala S des Ciphertextes festgelegt.

Anschließend werden SK und PK erzeugt und der Encryptor, Evaluator und Decryptor. Dieses Verfahren ähnelt sich wie zuvor bei dem BFV Algortihmus (siehe Codelisting 3.2). Bei CKKS ist zusätzlich noch ein RelinKey festzulegen (siehe dazu [*Microsoft SEAL (release 3.6)* 2020]). Der Key wird für das verschieben auf unterschiedliche Level benötigt (Codelisting 3.9 Zeile 3 und 7).
In Zeile 1 - 3 des Codelisting 3.8 werden die Werte x in einem Vektor gespeichert, mit denen gerechnet werden soll. Im Vergleich zu BFV hat sich beim Erstellen des Plaintextes für die Werte x nur verändert, dass die Skala S mit einberechnet wird (siehe Codelisting

```
1 vector<double> input;
2 input.push_back(2.5);
3 input.push_back(3.78);
4
5 Plaintext plain_coeff_pi;
6 encoder.encode(3.14159265, s, plain_coeff_pi);
```

Listing 3.8: Gleitkommazahlen in Plaintext Space verschlüsseln auf Basis von CKKS

3.8 Zeile 5 - 6).
In Codelisting 3.9, Zeilen 11-12 wird der Berechnungsprozess durchgeführt. Zunächst wird die Zeit vor der kompletten Berechnung gemessen (Codelisting 3.9 Zeile 2). Anschließend das Quadrat von x ermittelt (Codelisting 3.9 Zeile 3). Da mit CKKS gerechnet wird, wird das Ergebnis linearisiert und um ein Level, wie bereits in diesem Kapitel erklärt, nach unten geschoben. Anschließend wird das selbe mit $\pi * x$ getan. Da sich der Wert der Multiplikation aber ein Level oberhalb der vorherigen Multiplikation befindet, wird der Wert ebenfalls ein Level nach unten verschoben (Codelisting 3.9 Zeile 8). Da sich im Anschluss beide Werte auf dem selben Level befinden können sie multipliziert werden (Codelisting 3.9 Zeile 10-12). Die restlichen Zeilen 15-19 verhalten sich wie im Codelisting von A.1, nur dass das Ergebnis wieder in einen Vektor geschrieben werden muss (Codelisting 3.8 Zeile 19).
In dem Beispiel wurde mit den Zahlen 2,5 und 3,78 gerechnet (siehe Codelisting 3.8 Zeile 2-3). Bei der aktuellen Version [*Microsoft SEAL (release 3.6)* 2020] ergibt sich für die Berechnung der Wert folgende Zahlen als Ergebnis heraus:

- $\pi * 2.5^3 = 49.087385146$
  Erwartet: 49.087385212
- $\pi * 3.78^3 = 169.677896499$
  Erwartet: 169.677896742

```
1  Ciphertext x_square_2_encrypted;
2  double cpu_start_time = clock();
3  evaluator.square(x_encrypted, x_square_2_encrypted);
4  evaluator.relinearize_inplace(x_square_2_encrypted, relin_keys);
5  evaluator.rescale_to_next_inplace(x_square_2_encrypted);
6  Ciphertext x_encrypted_multiply_coeff_pi;
7  evaluator.multiply_plain(x_encrypted, plain_coeff_pi, x_encrypted_multiply_coeff_pi);
8  evaluator.rescale_to_next_inplace(x_encrypted_multiply_coeff_pi);
9  Ciphertext x_square_2_mul_x_mul_pi;
10 evaluator.multiply(
11 x_square_2_encrypted, x_encrypted_multiply_coeff_pi, x_square_2_mul_x_mul_pi
12 );
13 double cpu_end_time = clock();
14
15 Plaintext plain_result;
16 decryptor.decrypt(x_square_2_mul_x_mul_pi, plain_result);
17 vector<double> result;
18 encoder.decode(plain_result, result);
19 print_vector(result, 2, 7);
```

Listing 3.9: Berechnung von $\pi * x^3$ mit CKKS

Bei den Ergebnissen ist gut zu erkennen, dass sich bereits bei der sechsten Nachkommastelle Fehler ergeben.

Um den CKKS Algorithmus einschätzen zu können. bzgl. Performance und Korrektheit, wird ähnlich wie in Kapitel 3.1 1.000 Additionen ausgeführt (siehe 3.10). Dabei wird auf das zuvor errechnete Ergebnis aus diesem Kapitel (49.087385212 und 169.677896742) um die Zahl i des Zählers aus der for-Schleife addiert (siehe Codelisitng 3.10 Zeile 2-7). Die Laufzeit wird vor und nach den Durchläufen ermittelt. Anschließend werden die Durchläuft auf 10.000 erhöht. Bei der mehrfachen Addition von 0,1 ist aufgefallen, dass die Ergebnisse weiterhin einen Fehler ab der sechsten Nachkommastelle behalten. Eine Addition im CKKS Schema erhöht den Fehler nicht. In Tabelle 3.3 ist zu sehen, dass das CKKS Schema eine gute Laufzeit hat und 1.000 Addition im Schnitt unter einer Sekunde berechnet. Ähnlich wie bei dem BFV Schema (siehe Kapitel 3.1) steigert sich die Durchlaufzeit um den Faktor zehn, wenn die Anzahl der Durchläufe ebenfalls mit dem Faktor zehn erhöht wird.
Im Allgemeinen ist zu sagen, dass die der CKKS Algorithmus auf Grund seiner Laufzeit, wie der BFV Algorithmus, zur Anwendung eigenen würden. Negativ ist aufgefallen, dass der

```
double cpu_start_time = clock();
for (int i = 0; i < 1000; i++){
        Plaintext i_plain;
        encoder.encode(0.1, scale, i_plain);
        evaluator.mod_switch_to_inplace(i_plain, last_parms_id);
        evaluator.add_plain_inplace(encrypted_result, i_plain);
}
double cpu_end_time = clock();
```

Listing 3.10: Messen der Berechnungsdauer von $\pi * x^3$ mit CKKS bei 1.000 Wiederholungen

CKKS Algorithmus Nachkommastellen Fehler hat. Das ist bei einer Gleitkommaberechnung in der Computertechnik jedoch nicht ungewöhnlich.

Durchlauf	Zeit (1.000 Berechnungen)	Zeit (10.000 Berechnungen)
1	0.104399 sec	1.24231 sec
2	0.097456 sec	0.976632 sec
3	0.09741 sec	0.970339 sec
4	0.097069 sec	0.971901 sec
5	0.098252 sec	0.961099 sec
Mittelwert	**0.098917 sec**	**1.02445 sec**

Tabelle 3.3: Berechnungsdauer homomorph verschlüsselte Gleichung mit dem CKKS Schema

3.3 Datenanalyse mit und ohne homomorph verschlüsselten Daten

In diesem Kapitel wollen wir der Frage nachgehen, wie sehr sich homomorphe Verschlüsselung bereits für Operationen auf Kundendaten eignet. Dazu legen wir einen Datensatz von 30 Personenbezogene Daten (siehe Abbildung 3.2), bzw. voller Datensatz unter Tabelle A.1 in MariaDB an. Die selben Daten sollen in einen Datentypen von C++ eingefügt werden und anschließend mit dem Transpiler von Google in deren homomorph Verschlüsselte Bibliothek übersetzt werden. Auf diese Weise kann einfacher C++ Code verwendet werden, um sehr komplexe homomorphe Algorithmen auszuführen. Auf beiden Datensätzen wird ein Test durchgeführt, um zu evaluieren, wie sich homomorph verschlüsselte Daten im Vergleich zu einer üblichen Anwendung verhält. Der Datensatz erhält eine ID, den Vor- und Nachnamen, sowie eine Postleitzahl. Die Postleitzahlen sind dabei auf 76131 - 76137 beschränkt. Zunächst möchten wir herausfinden wie viele Personen in dem Datensatz in

```
MariaDB [he]> select * from costumer;
+----+-----------+-------------+-------+
| id | firstname | lastname    | place |
+----+-----------+-------------+-------+
|  1 | Peter     | Marx        | 76131 |
|  2 | Jason     | Cochran     | 76133 |
|  3 | Abdullah  | Gilmore     | 76135 |
|  4 | Bradleigh | Poole       | 76135 |
|  5 | Caio      | Fitzgerald  | 76137 |
|  6 | Zubair    | Cano        | 76137 |
|  7 | Cydney    | Lu          | 76131 |
```

Abbildung 3.2: Ausschnitt Datenbankstruktur MariaDB

einem bestimmten Ort wohnen.
Dazu führen wir die Berechnungen in SQL Syntax durch und anschließend wird der Ablauf in C++ programmiert, sodass der Code transpiliert werden kann. Um die Daten später vergleichen zu können, werden die Berechnungen auf dem gleichen Computer durchgeführt wie zu beginn des Kapitel 3 bereits erwähnt.

3.3.1 Datenanalyse mit MariaDB als Grundlage

Wir möchten in SQL Syntax herausfinden, wie viele Personen in dem Stadtteil mit der Postleitzahl *76133* wohnen. Dazu wird die Suche fünf mal ausgeführt und der Mittelwert der Operationen berechnet(siehe Abbildung 3.3). Die Zeit die eine übliche Datenbank

```
MariaDB [he]> SELECT * FROM costumer WHERE place = "76133";
+----+-----------+----------+-------+
| id | firstname | lastname | place |
+----+-----------+----------+-------+
|  2 | Jason     | Cochran  | 76133 |
| 10 | Chance    | Merritt  | 76133 |
| 12 | Sanna     | Shea     | 76133 |
| 22 | Shib      | Dyln     | 76133 |
| 27 | Edie      | Woods    | 76133 |
| 30 | Claudia   | Corona   | 76133 |
+----+-----------+----------+-------+
6 rows in set (0.004 sec)
```

Abbildung 3.3: Längster Durchlauf bei der Suche nach der Postleitzahl

benötigt beläuft sich dabei auf folgende:

Anzahl Durchlauf	Zeit
1	0.004 sec
2	0.001 sec
3	0.000 sec
4	0.001 sec
5	0.001 sec
Mittelwert	**0.0014 sec**

Tabelle 3.4: Berechnungsdauer in MariaDB

Dabei muss beachtet werden, dass das relationale Datenbankmodell, auf welches hier in Form von MariaDB zurückgegriffen wird, das erste mal im Juni 1970 in einem Paper beschrieben worden ist [Codd 1970]. Datenbanken können somit auf eine Technik zurückgreifen die in den letzten 50 Jahren stark optimiert und verbessert wurde. Deswegen

ist stark davon auszugehen, dass die Durchläufe zwei bis fünf in Tabelle 3.4 durch Caching optimiert sind.

3.3.2 Datenanalyse mit Googles C++ Transpiler

Anschließend werden die Daten in C++ übertragen, sodass mit Hilfe von Googles Transpiler erneut fünf Durchgänge der 30 Personen berechnet werden kann. Zusätzlich soll der Suchlauf um Datensätze mit fünf und zehn Personen ergänzt werden. Dadurch ist ein besseres Gefühl für die Performance des Transpiler gegeben.
In dieser Arbeit wird sich zunächst darauf beschränkt zu zählen, wie viele Personen einer bestimmten Postleitzahl in dem Datensatz vorhanden sind. Sollte dieser Versuch erfolgreich sein, ist davon auszugehen, dass weitere komplexe Versuche, z. B. Rückgabe von Namen aller Personen einer gewissen Postleitzahl, möglich sind. Dabei sind die Daten bei der Rückgabe verschlüsselt.
Mit dem Transpiler von Google lässt sich eine möglich Client Server Anwendung simulieren (siehe Abbildung 3.4). Diese Simulation soll im folgendem auf einem einzigem Computer

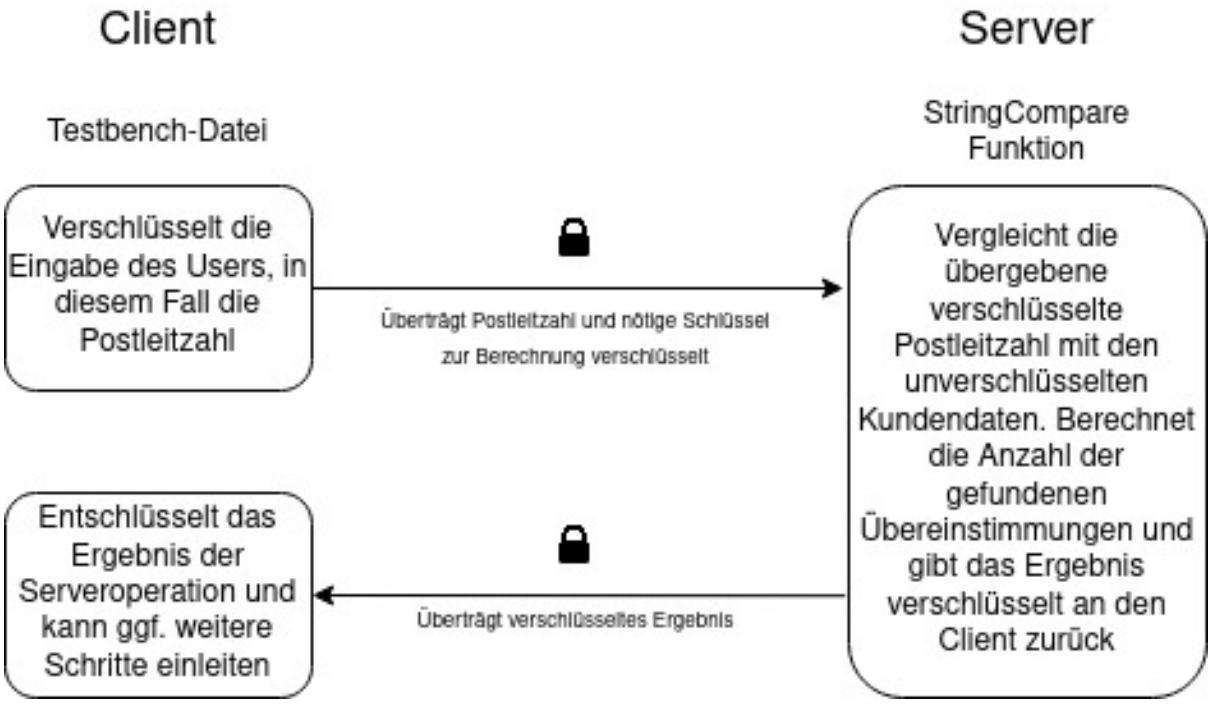

Abbildung 3.4: möglicher Ablauf einer Client Server Anwendung für homomorphe Verschlüsselung

getestet werden. Dazu wird der Transpiler, wie in Kapitel 2.4 beschrieben, mit den fünf

Phasen verwendet. Der Programmierer muss für seinen eigenen Test die Testbench-Datei (siehe Abbildung 3.4) anlegen. Die Testbench-Datei für dieses Projekt ist unter dem Codelisting A.3 gegeben. Wie in Kapitel 2.4 beschreiben wird der Code vom Transpiler entsprechend vorbereitet. Der Programmierer muss, wie bei dem BFV und CKKS Schema (siehe Kapitel 3.1 und Kapitel 3.2), die Parameter zur Erstellung der Schlüssel in der Testbench-Datei selber wählen (siehe Codelisting 3.11 Zeile 1-15).

Ähnlich wie bei dem BFV und CKKS Schema gibt es auch einen Plaintextspace bzw.

```
1  constexpr int kMainMinimumLambda = 120;
2
3  //...
4
5  TFheGateBootstrappingParameterSet* params =
6  new_default_gate_bootstrapping_parameters(kMainMinimumLambda);
7
8  // generate a random key
9  // Note: In real applications,
10 // a cryptographically secure seed needs to be used.
11 uint32_t seed[] = {314, 1592, 657};
12 tfhe_random_generator_setSeed(seed, 3);
13 TFheGateBootstrappingSecretKeySet* key =
14 new_random_gate_bootstrapping_secret_keyset(params);
15 const TFheGateBootstrappingCloudKeySet* cloud_key = &key->cloud;
16
17 std::string plaintext1(input1);
18
19 // Encrypt data
20 auto ciphertext1 = FheString::Encrypt(plaintext1, key);
```

Listing 3.11: Testbench-Datei - Erstellung der nötigen Schlüssel und Verschlüsselung der Eingabe

Ciphertexte. Zur Verschlüsselung des Inputs benötigt die FHE-Bibliothek von Google ebenfalls die Schlüsselpaare, um den Ciphertext zu erstellen (siehe Codelisting 3.11 Zeile 20). Der Verschlüsselte Speicherbereich, indem das Ergebnis gespeichert werden soll wird als *cipher_result* definiert (siehe Codelisting 3.12 Zeile 2). Der Transpiler hat wie bereits in diesem Kapitel erwähnt, die Schritte durchgeführt (siehe Kapitel 2.4), die dafür sorgen, dass der Code als XLS IR interpretiert werden kann. Daher wird die Funktion *stringCompare* der XLS IR Funktion *XLS_CHECK_OK* (siehe Codelisting 3.12 Zeile 4-6) übergeben.

Der Aufruf der Funktion *XLS_CHECK_OK* ruft die stringCompare Funktion auf und führt die homomorph verschlüsselte Berechnung durch. Dabei benötigt die Funktion die zuvor erwähnt und definierten Variablen *ciper_result*, um das Ergebnis verschlüsselt zurückzugeben, die verschlüsselte Eingabe *ciphertext1* und die benötigten Schlüssel, wie in Kapitel 2.1 beschreiben.

```
1 // Perform string compare
2 FheInt cipher_result(params);
3
4 XLS_CHECK_OK(stringCompare(
5         cipher_result.get(), ciphertext1.get(), cloud_key
6 ));
```

Listing 3.12: Testbench-Datei - Aufruf der stringCompare Funktion

In der C++ Datei, in der die *stringCompare*-Funktion definiert ist (siehe Codelisting A.4 und Abbildung 3.4) wird ermittelt, wie oft die gesuchte Postleitzahl in den Kundendaten (siehe Tabelle A.1) vorhanden ist. Dabei ist zu erwähnen, dass die Kundendaten hier nicht verschlüsselt vorliegen. Das hat den Grund, dass Operationen mit dem Transpiler bisher nur experimentell möglich sind [*Fully Homomorphic Encryption (FHE)* 2021]. Die Berechnungen sind bisher sehr langsam und nicht optimiert. Der Einfachheit halber ist in diesem Beispiel nur die gesuchte Postleitzahl verschlüsselt und wird mit dem unverschlüsseltem Kundenstamm abgeglichen. Es ist möglich die Kundendaten in der Testbench-Datei (Codelisiting A.3) zu verschlüsseln und ebenfalls verschlüsselt der *stringCompare*-Funktion zu übergeben. Dabei würde sich die Berechnungsdauer jedoch so stark erhöhen, dass in diesem Beispiel darauf verzichtet wurde.
Die Daten der Kunden sind in einem char Array fester Größe abgelegt (siehe Codelisting A.4 Zeile 7-12). Die variabel wählbar zu suchende Postleitzahl wird der Funktion übergeben (siehe Zeile 3). Anschließend wird jede Speicherstelle des char Arrays miteinander verglichen (siehe Zeile 15-23 und Zeile 28-36). Sollte eine Postleitzahl einer Person mit der gesuchten übereinstimmen, wird eine Zählvariable um jeweils eins erhöht und der Wert am Ende der Funktion zurückgegeben (siehe Zeile 21,34 und 38).
Der Transpiler kann mit Hilfe des Tools Bazel [*Fully Homomorphic Encryption (FHE)* 2021] getestet werden, welches direkt von Google entwickelt wird. Bazel ist ein vielseitiges Tool und kann einige Aufgaben übernehmen, z. B. Code-Generation, Resource-Generation, Compiliation, Packaging, Testing, Integration Testing oder Deploying. In diesem Fall übernimmt Bazel die Aufgabe, den Code der Testbench-Datei auszuführen. Ein erfolgreicher

Durchlauf im Command Line Interface kann in Abbildung 3.5 betrachtet werden.

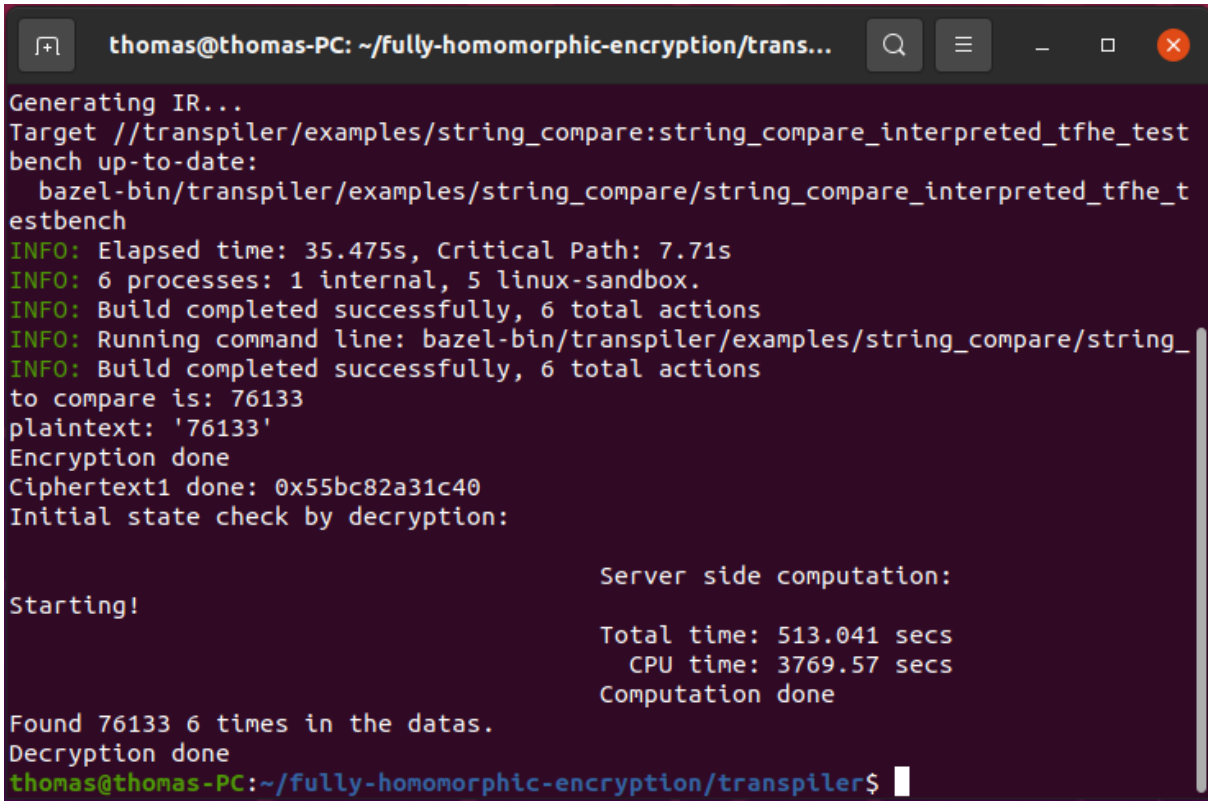

Abbildung 3.5: Durchlauf bei der Suche nach der Anzahl an einer Postleitzahl - homomorph Verschlüsselt.

Durchlauf	Zeit (Personen: 5)	Zeit (Personen: 10)	Zeit (Personen: 30)
1	238.302 sec	347.117 sec	3753.72 sec
2	237.515 sec	354.446 sec	3687.46 sec
3	242.467sec	349.511 sec	3694.84 sec
4	247.45 sec	355.433 sec	3805.63 sec
5	233.157 sec	354.987 sec	3748.48 sec
Mittelwert	**239.778 sec**	**352.2988 sec**	**3738.03 sec**

Tabelle 3.5: Berechnungsdauer homomorph verschlüsselte Daten

In Tabelle 3.4 und Tabelle 3.5 ist festzustellen, dass die Verarbeitung auf Serverseite von homomorph verschlüsselten Daten sehr viel länger dauert, als bei einer relationalen Datenbank. Die Laufzeit ist 2.500.000 mal länger. Zu erwähnen ist auch, dass die Zeit sich bei der Suche zwischen fünf und zehn Personen nicht verdoppelt. Auf verdreifacht sich die Zeit bei dem Vergleich von fünf bzw. 30 Personen im Datensatz nicht, sondern verzehnfacht sich sogar. Damit ist anzunehmen, dass der Transpiler polynomiale Laufzeit hat. Google beschreibt die Laufzeit der FHE-C++ Bibliothek, auf die der Transpiler transpiliert, selbst als nicht praktikabel:
"While it could be deployed in practice, the run-times of the FHE-C++ operations are likely to be too long to be practical at this time. " [*Fully Homomorphic Encryption (FHE)* 2021]
Dennoch ist anzumerken, dass komplizierte Berechnungen, bzw. eine einfache Form von Data Insights durchgeführt wurde. Damit ist festzuhalten, dass die homomorphe Verschlüsselung durchaus in der Lage ist solche Operationen durchzuführen. Es zeigt sich aber auch, dass der Transpiler von Google noch nicht in der Lage ist, die Anwendung in die Praxis zu übertragen, dazu fehlt die Performance der Operation. Angenommen es gäbe einen Datensatz mit über 1 Millionen Kunden, was nicht ungewöhnlich ist, würde die Laufzeit so lange dauern, dass es nicht akzeptabel wäre.

3.4 Zusammenfassung der Ergebnisse aus den praktischen Tests

In diesem Kapitel, werden die Erkenntnisse des Kapitel 3 kurz zusammengefasst.
In Kapitel 3.1 und 3.2 wurde gezeigt, dass beide Algorithmen sehr effektiv arbeiten. Die Zeiten der Berechnungen sind so schnell, das es denkbar ist, die Algorithmen schon heute in produktiv Code einzubeziehen. BFV eignet sich auf Grund der Geschwindigkeit und der Genauigkeit zum Berechnen von verschlüsselten Ganzzahlen. Der einzige Nachteil, der

sich bei CKKS herausgestellt hat, ist dass die Ergebnisse kleinere Fehler haben. Sollte aber auf diesen negativen Nebeneffekt verzichtet werden können, eignet sich CKKS sehr gut zur verschlüsselten Berechnung von Gleitkommazahlen. Zusätzlich ist anzumerken, dass die meisten Gleitkommaberechnungen in der IT meist nicht korrekt sind.
Bei der Datenanalyse in Kapitel 3.3 hat sich gezeigt, dass der Transpiler von Google noch nicht sehr effizient, im Vergleich zu konventionellen Methoden arbeitet. Die Laufzeiten für nur 30 Personen in einem Datensatz sind viel zu zeitintensiv. Dennoch hat sich aber in dem Kapitel 3.3 gezeigt, dass es durchaus möglich ist, einfache Datenanalyse mit verschlüsselten Daten durchzuführen.

Kapitel 4

Fazit

Mit dieser Projektarbeit sollte evaluiert werden, ob homomorphe Verschlüsselung bereits praktische Anwendung im Alltag der Informationstechnologie finden kann. Dabei werden systematisch drei Anwendungsfälle getestet (siehe Kapitel 1.2). Im laufe der Arbeit hat sich herausgestellt, dass die gesetzten Ziele mit bereits vorhanden Bibliotheken umzusetzen sind. Die Bibliotheken bringen bereits die Kernfunktionen des Algorithmus, z. B. für die Erstellung der Schlüssel zum verschlüsseln mit (siehe Kapitel 2). Dabei sind vor allem die Bibliotheken von Microsoft und Google in dieser Arbeit als relevant herausgearbeitet worden (siehe Kapitel 2.4).
Zuerst wurde die Berechnung von Ganzzahlen (siehe Kapitel 3.1) und Gleitkommazahlen (siehe Kapitel 3.2) evaluiert. Es hat sich bei der Evaluierung herausgestellt, dass Ganzzahlberechnungen durchaus praxistauglich sind. Die Ergebnisse sind immer korrekt und auch schnell durchgeführt. Der Vorteil bei der Ganzzahlberechnung liegt hier ganz klar bei den korrekten Ergebnissen, welche bei der homomorphen Verschlüsselung nicht selbstverständlich sind (siehe Kapitel 3.2).
Berechnungen mit Gleitkommazahlen sind noch nicht vollkommen praxistauglich. Sie lassen sich zwar schnell und performant ausführen, liefern aber wegen des zu Grunde liegenden Algorithmus keine korrekte Ergebnisse. Dieser Fehler kann jedoch auf die Gleitkommaarithmetik zurückgeführt werden. Die Zahlen weichen vom erwarteten Ergebnis ab. Ebenfalls lassen sich nur eine gewisse Anzahl von Berechnungen durchführen, da der Speicherplatz der Verschlüsselung begrenzt ist.
Nach der einfachen Addition und der Berechnung der Gleitkommazahlen hat sich in der Arbeit ebenfalls gezeigt, dass es sogar möglich ist einfache Strings miteinander zu vergleichen (siehe Kapitel 3.3). Damit ist es auf einfachste Art und Weise bewiesen, dass sich die homomorphe Verschlüsselung praktisch sogar für einfache Data Insights

Operationen eignet. Bei den Tests hat sich herausgestellt, dass die Berechnungen auf String Operationen sehr langsam sind. Mit der in dieser Arbeit verwendeten Hardware [1], benötigt eine homomorph verschlüsselte Operation zum Stringvergleich fast 2.500.000 mal länger als eine übliche Technik in der Informatik (siehe Kapitel 3.3). Aus diesem Grund ist es zwar möglich solche Berechnungen durchzuführen, sie sind aber in der Praxis noch nicht effizient im Bezug auf die Geschwindigkeit, um eingesetzt zu werden.
Laut dem Security-Paper der Homomorphic Encryption Organisation [Chase u. a. 2017] gilt die homomorphe Verschlüsselung mit aktuellen Kenntnissen als sicher. Laut der Organisation und einigen Papern [Gentry 2009], [Cheon u. a. 2016], gilt das auch für die Post Quanten Kryptographie.

[1] CPU: AMD Ryzen 5 3500U, RAM: 14,6 GB, Festplattenart: SSD

Kapitel 5

Ausblick

Das Thema homomorphe Verschlüsselung steht relativ am Anfang. Zwar ist es wie im Kapitel 3 beschrieben schon möglich etliche Berechnungen durchzuführen, das Thema an sich hat noch viel Potenzial.
In Zukunft könnte es möglich sein, dass z. B. die Daten der Produkte von Microsoft Office 365, oder Google Dienste, vollständig homomorph Verschlüsselt verarbeitet werden. Das hat den Vorteil, dass die Anbieter die Daten voll verschlüsselt auf ihren Servern verarbeiten können und dadurch beispielsweise die DSGVO einfacher umzusetzen ist.
In der Arbeit wurden nur die Bibliotheken vom Microsoft und Google getestet. Es ist aber auch zu erwähnen, dass es weit aus mehr private und kommerziell geführte Bibliotheken gibt. Ein Beispiel ist IBM [Masters u. a. 2020], die sich ebenfalls ausgiebig mit dem Thema beschäftigt haben.
Zusätzlich ist zu erwähnen, dass Google in einem Blogpost angekündigt hat, weitere Innovationen in diesem Bereich zu veröffentlichen: *"We will continue to invest and lead the privacy-preserving technology field by publishing new work, and open-sourcing it for everyone to use at scale - and we're excited to continue this practice by sharing this latest advancement with developers everywhere."* [Guevara 2021]. Durch diese Aussage ist zu erwarten, dass in der nächsten Zeit der Transpiler aber auch Googles FHE-Bibliothek weiterentwickelt wird und das Thema von deren Seite nicht abgeschlossen ist. Somit ist mit neuen Innovationen zu rechnen.
Google wagt sogar eine noch weitaus interessanter Prognose abzugeben und zwar dass die homomorphe Verschlüsselung in bereits zehn Jahren in der Forschung von Genen helfen kann: *„In the next 10 years, FHE could even help researchers find associations between specific gene mutations by analyzing genetic information across thousands of encrypted samples and testing different hypotheses to identify the genes most strongly associated*

with the diseases they're studying." [Guevara 2021]. Die homomorphe Verschlüsselung soll laut Google das nötige Vertrauen schaffen, dass Menschen ihre genetische Informationen verschlüsselt teilen. Dabei hätte das Unternehmen, welches die genetischen Daten bekommt, auf Grund der Verschlüsselung keine Chance die menschlichen Eigenschaften selber zu lesen. Der entscheidende Punkt ist aber, dass Google es für möglich hält, dass ein großer Pool an genetischen Daten entsteht - und das weltweit.

Anhang A

Anhang

id	firstname	lastname	place	id	firstname	lastname	place
1	Peter	Marx	76131	16	Kobe	Higgs	76137
2	Jason	Cochran	76133	17	Lauren	Worthington	76135
3	Abdullah	Gilmore	76135	18	Peter	Magnus	76131
4	Bradleigh	Poole	76135	19	Coy	Karen	76131
5	Caio	Fitzgerald	76137	20	Tihomira	Petra	76135
6	Zubair	Cano	76137	21	Shib	Dylan	76137
7	Cydney	Lu	76131	22	Shib	Dyln	76133
8	Jim	Whitefield	76131	23	Rakesh	Sanna	76131
9	Tonya	Milne	76135	24	Coral	Norton	76137
10	Chance	Merritt	76133	25	Lauren	Christensen	76137
11	Amarah	Seymour	76137	26	Axel	Stein	76131
12	Sanna	Shea	76133	27	Edie	Woods	76133
13	Kameron	Redman	76135	28	Tasha	Whelan	76131
14	Kornelia	Shannon	76131	29	megan	Craig	76137
15	Riccardo	Stout	76131	30	Claudia	Corona	76133

Tabelle A.1: Datensatz zur Vergleichanalyse SQL und Google TFHE

```
1  //setze Parameter siehe Kapitel 2.1
2  EncryptionParameters parms(scheme_type::bfv);
3  size_t poly_modulus_degree = 4096;
4  parms.set_poly_modulus_degree(poly_modulus_degree);
5  parms.set_coeff_modulus(CoeffModulus::BFVDefault(poly_modulus_degree));
6  parms.set_plain_modulus(1024);
7  // der Context validiert die Eingabe der Parameter
8  SEALContext context(parms);
9
10 //erzeuge Public Private Key
11 KeyGenerator keygen(context);
12 SecretKey secret_key = keygen.secret_key();
13 PublicKey public_key;
14 keygen.create_public_key(public_key);
15
16 Encryptor encryptor(context, public_key);
17 Evaluator evaluator(context);
18 Decryptor decryptor(context, secret_key);
19
20 //Zahlen die addiert werden sollen, Hexadezimal
21 int x = 8;
22 int y = 19;
23 Plaintext x_plain(to_string(x));
24 Plaintext y_plain(to_string(y));
25
26 Ciphertext x_encrypted;
27 Ciphertext y_encrypted;
28 Ciphertext x_plus_y_encrypted;
29 //Verschlüsselung der Zahlen x und y
30 encryptor.encrypt(x_plain, x_encrypted);
31 encryptor.encrypt(y_plain, y_encrypted);
32
33 // der Evaluator addiert zwei Ciphertexte und speichert das Ergebniss in einem drit
34 evaluator.add(x_encrypted, y_encrypted, x_plus_y_encrypted);
35
36 Plaintext decrypted_result;
37 decryptor.decrypt(x_plus_y_encrypted, decrypted_result);
38 cout << "0x" << decrypted_result.to_string() << endl;
```

Listing A.1: Einfache homomorphe verschlüsselte Addition mit Microsoft SEAL auf Basis von BFV

```
EncryptionParameters parms(scheme_type::ckks);
size_t poly_modulus_degree = 8192;
parms.set_poly_modulus_degree(poly_modulus_degree);
parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 60, 40, 40, 60 }));
double s = pow(2.0, 40);
SEALContext context(parms);

// Generiere Encryptor, Evaluator, Decryptor, siehe dazu Codebeispiel BFV

CKKSEncoder encoder(context);

vector<double> input;
input.push_back(2,5);
input.push_back(3,78);

//Plaintext Space für den Koeffizienten Pi wird erzeugt
Plaintext plain_coeff_pi;
encoder.encode(3.14159265, s, plain_coeff_pi);

Plaintext x_plain;
encoder.encode(input, s, x_plain);
Ciphertext x_encrypted;
encryptor.encrypt(x_plain, x_encrypted);

// um x^3 zu berechnen, errechnen wir erst das Quadrat von x und reliniarisieren
// anschließend berechnen wir Pi * x und schieben das Produkt auf den selben Level
// da beide Produkte auf dem selben Level sind, multiplizieren wir
Ciphertext x_square_2_encrypted;
evaluator.square(x_encrypted, x_square_2_encrypted);
evaluator.relinearize_inplace(x_square_2_encrypted, relin_keys);
evaluator.rescale_to_next_inplace(x_square_2_encrypted);
Ciphertext x_encrypted_multiply_coeff_pi;
evaluator.multiply_plain(x_encrypted, plain_coeff_pi, x_encrypted_multiply_coeff_pi);
evaluator.rescale_to_next_inplace(x_encrypted_multiply_coeff_pi);
Ciphertext x_square_2_mul_x_mul_pi;
evaluator.multiply(
x_square_2_encrypted, x_encrypted_multiply_coeff_pi, x_square_2_mul_x_mul_pi
);

Plaintext plain_result;
decryptor.decrypt(x_square_2_mul_x_mul_pi, plain_result);
vector<double> result;
encoder.decode(plain_result, result);
print_vector(result, 2, 7);
```

Listing A.2: Komplexe homomorphe verschlüsselte Multiplikation mit Microsoft SEAL auf Basis von CKKS

```
constexpr int kMainMinimumLambda = 120;
int main(int argc, char** argv) {
        if (argc < 2) {
                std::cerr << "Usage: string_compare_fhe_testbench plz" << std::endl;
                return 1;
        }

        std::string input1 = argv[1];
        input1.resize(MAX_LENGTH, '\0');
        std::cout << "to compare is: " << input1 << std::endl;

        TFheGateBootstrappingParameterSet* params =
        new_default_gate_bootstrapping_parameters(kMainMinimumLambda);

        // generate a random key Note: In real applications,
        // a cryptographically secure seed needs to be used.
        uint32_t seed[] = {314, 1592, 657};
        tfhe_random_generator_setSeed(seed, 3);
        TFheGateBootstrappingSecretKeySet* key =
        new_random_gate_bootstrapping_secret_keyset(params);
        const TFheGateBootstrappingCloudKeySet* cloud_key = &key->cloud;

        std::string plaintext1(input1);
        // Encrypt data
        auto ciphertext1 = FheString::Encrypt(plaintext1, key);
        std::cout << "Encryption done" << std::endl;
        std::cout << "Ciphertext1 done: " << ciphertext1.get() << std::endl;

        std::cout << "\t\t\t\t\tServer side computation:" << std::endl;
        std::cout << "Starting!" << std::endl;

        // Perform string compare
        FheInt cipher_result(params);
        XLS_CHECK_OK(stringCompare(
        cipher_result.get(), ciphertext1.get(), cloud_key
        ));

        std::cout << "\t\t\t\t\tComputation done" << std::endl;
        std::cout << "Found " << input1 << " " << cipher_result.Decrypt(key)
        << " times in the datas. \n";
}
```

Listing A.3: Testbench-Datei für Googles C++ Transpiler

```
#include "string_compare.h"
#pragma hls_top
int stringCompare(char firstPlz[MAX_LENGTH]){
        int matches = 0;
        int count = 0;

        char PeterMarx[6] = "76131";
        char JasonCochran[6] = "76133";
        char AbdullahGilmore[6] = "76135";
        //..having the other 25 persons declared here
        char MeganCraig[6] = "76137";
        char ClaudiaCorona[6] = "76133";

        #pragma hls_unroll yes
        for (int i = 0; i <= MAX_LENGTH; ++i) {
                if(firstPlz[i] == PeterMarx[i]){
                        count++;
                }
        }
        if (count >= 5) {
                matches++;
        }
        count = 0;

        //having multiple times the same Code here again for all other persons

        #pragma hls_unroll yes
        for (int i = 0; i <= MAX_LENGTH; ++i) {
                if(firstPlz[i] == ClaudiaCorona[i]){
                        count++;
                }
        }
        if (count >= 5) {
                matches++;
        }
        count = 0;

        return matches;
}
```

Listing A.4: Analyse der Anzahl von Postleitzahlen in nicht transpiliertem C++ Code

Literaturverzeichnis

Albrecht, Martin u. a. [2018]. *Homomorphic Encryption Security Standard*. Techn. Ber. Toronto, Canada: HomomorphicEncryption.org [siehe S. 6–10].

Chase, Melissa u. a. [2017]. *Security of Homomorphic Encryption*. Techn. Ber. Redmond WA, USA: HomomorphicEncryption.org [siehe S. 11, 36].

Cheon, Jung Hee u. a. [2016]. *Homomorphic Encryption for Arithmetic of Approximate Numbers*. Cryptology ePrint Archive, Report 2016/421. `https://eprint.iacr.org/2016/421`. [Besucht am 06. 09. 2021] [siehe S. 11, 36].

Codd, Edgar F. [1970]. *A Relational Model of Data for Large Shared Banks*. Techn. Ber. IBM Research Laboratory, San Jose, California [siehe S. 28].

Drehling, Wilhelm [2021]. *Reingefallen*. Seite 60ff. Heise Medien GmbH & Co. c't 07/2021 [siehe S. 11].

Drehling, Wilhelm und Sylvester Tremmel [2021]. *Post-Quanten-Portfolio*. Seite 68ff. Heise Medien GmbH & Co. c't 16/2021 [siehe S. 12].

Fully Homomorphic Encryption (FHE) [2021]. `https://github.com/google/fully-homomorphic-encryption`. [Besucht am 06. 09. 2021] [siehe S. 13, 14, 31, 33].

Gentry, Craig [2009]. »A Fully Homomorphic Encryption Scheme«. dissertation. Standford University [siehe S. 2, 36].

Google [o. D.] `https://google.github.io/xls/`. [Besucht am 06. 09. 2021] [siehe S. 14].

Grävemeyer, Arne [2021]. *Kryptoagil gegen hackende Quantencomputer*. Seite 62ff. Heise Medien GmbH & Co. c't 16/2021 [siehe S. 12].

Guevara, Miguel [2021]. *Our latest updates on Fully Homomorphic Encryption*. `https://developers.googleblog.com/2021/06/our-latest-updates-on-fully-homomorphic-encryption.html`. [Besucht am 06. 09. 2021] [siehe S. 13, 37, 38].

iDASH secure genome analysis competition 2018: blockchain genomic data access logging, homomorphic encryption on GWAS, and DNA segment searching [2020]. `https://doi.org/10.1186/s12920-020-0715-0`. [Besucht am 06. 09. 2021] [siehe S. 6].

Kuobin, Dai [2011]. »PGP E-Mail Protocol Security Analysis and Improvement Program«. In: *2011 International Conference on Intelligence Science and Information Engineering*. DOI: `10.1109/ISIE.2011.144` [siehe S. 1].

Lattigo v2.2.0 [Juli 2021]. Online: `http://github.com/ldsec/lattigo`. EPFL-LDS [siehe S. 13].

Li, Baiyu und Daniele Micciancio [2020]. *On the Security of Homomorphic Encryption on Approximate Numbers*. Cryptology ePrint Archive, Report 2020/1533. `https://ia.cr/2020/1533` [siehe S. 13].

Masters, Oliver u. a. [Jan. 2020]. *Towards a Homomorphic Machine Learning Big Data Pipeline for the Financial Services Sector*. `https://eprint.iacr.org/2019/1113.pdf`. [Besucht am 06. 09. 2021] [siehe S. 37].

Microsoft [2021]. `https://www.microsoft.com/en-us/research/project/microsoft-seal/`. [Besucht am 08. 09. 2021] [siehe S. 3].

Microsoft SEAL (release 3.6) [Nov. 2020]. `https://github.com/Microsoft/SEAL`. Microsoft Research, Redmond, WA. [Besucht am 06. 09. 2021] [siehe S. 9–11, 13, 18, 22–24].

Regev, Oded [2005]. *The Learning with Errors Problem*. Techn. Ber. Blavatnik School of Computer Science, Tel Aviv University [siehe S. 12].

Rivest, Ronald L., L. Adleman und M. L. Dertouzos [1978]. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems* [siehe S. 1, 6, 11].

Tranquillus, C. Suetonius und Ed. Divus Julius Maximilian Ihm [o. D.] `http://data.perseus.org/citations/urn:cts:latinLit:phi1348.abo011.perseus-lat1:56.6`. [Besucht am 06. 09. 2021] [siehe S. 1].