Comenius-Gymnasium Deggendorf Abiturjahrgang 2023

# S E M I N A R A R B E I T

Rahmenthema des Wissenschaftspropädeutischen Seminars:

*Wissenschaftliche technische Oberstufe*

Leitfach: *WTO*

Thema der Arbeit:

## Homomorphe Verschlüsselung

Verfasser:
*Leon Adomaitis*

Kursleiter/in:
*Peter Mehrl / Stefanie Schneider*

Abgabetermin: *8. November 2022*

| **Bewertung** | Note | Notenstufe in Worten | Punkte | | Punkte |
|---|---|---|---|---|---|
| schriftliche Arbeit | | | | x 3 | |
| Abschlusspräsentation | | | | x 1 | |
| | | | | Summe: | |
| | | | | Gesamtleistung nach § 29 (7) GSO = Summe **: 2** (gerundet) | |

Datum und Unterschrift der Kursleiterin bzw. des Kursleiters

# Inhaltsverzeichnis

# 1 Motivation

Immer öfter werden Datensätze zum Verwalten in eine Public Cloud ausgelagert. Das geschieht jedoch oft auch mit sensiblen Daten, die auch personenbezogen sein können. Ein konkretes Beispiel sensibler Daten, wären persönliche medizinische Unterlagen. Jedoch könnte es sich bei sensiblen Daten, auch über vertrauliche Daten von Unternehmen handeln.[1]

In der Cloud werden diese Daten nicht nur abgespeichert, sondern auch verarbeitet. Dieser Prozess wird auch als Cloud Computing bezeichnet.[2]

Cloud Computing gewährt im Gegensatz zur lokalen Variante Vorteile wie beispielsweise eine *„Kostenreduzierung für Computer-Hardware“*[3], eine *„Kostenreduzierung für Software“*[4] und eine *„Verbesserte Performance“*[5]. Dennoch erweist sich dies als nicht unproblematisch. Denn um den Datenschutz zu gewährleisten, muss *„die Integrität und Vertraulichkeit der Datenverarbeitung“*[6] gewährleistet werden. Dies ist ein Punkt, der in vielen Fällen nicht gewährleistet werden kann.[7]

Um den Datenschutz dennoch gewährleisten zu können, ist es üblich, Daten zu verschlüsseln. Dies ist auch der Punkt, an dem häufig die Probleme entstehen, denn die meisten modernen Verschlüsslungen können für Berechnungen auf chiffrierten Daten nicht verwendet werden. Jedoch gibt es eine Verschlüsselungsart, die dies umgehen soll. Diese Kategorie, die Berechnungen auf verschlüsselten Daten erlaubt, wird homomorph genannt.[8]

Das Ziel dieser Arbeit ist es, eine grobe Einführung in die homomorphe Verschlüsselung zu geben. Ferner werden die zwei Unterscheidungskriterien der modernen Verschlüsselung

---

[1] Vgl. Lena Wiese / Daniel Homann / Tim Waage / Michael Brenner, »Homomorphe Verschlüsselung für Cloud-Datenbanken: Übersicht und Anforderungsanalyse«, in: Hanno Langweg / Michael Meier / Bernhard C. Witt / Delphine Reinhardt (Hrsg.), *SICHERHEIT 2018,* Bonn 2018, S. 221–234, hier: S. 221–222.
[2] Vgl. Thilo Weichert, »Cloud Computing und Datenschutz«, *Datenschutz und Datensicherheit - DuD*, 34, 2010, S. 679–687, hier: S. 679.
[3] Robert Vogel / Tarkan Koçoğlu / Thomas Berger, »Cloud Computing«, in: Robert Vogel / Tarkan Koçoğlu / Thomas Berger (Hrsg.), *Desktopvirtualisierung: Definitionen – Architekturen – Business-Nutzen Mit 35 Abbildungen und 16 Tabellen,* Wiesbaden 2010, S. 119–137, hier: S. 127.
[4] Ebd.
[5] Ebd.
[6] Weichert, Cloud Computing und Datenschutz, S. 680.
[7] Vgl. Wiese / Homann / Waage / Brenner, Homomorphe Verschlüsselung für Cloud-Datenbanken: Übersicht und Anforderungsanalyse, S. 221–222.
[8] Vgl. ebd., S. 222.

aufgezeigt und erläutert. Um ein besseres Verständnis zu bekommen, wie das funktioniert, wird am Ende das Konzept dieser Verschlüsselung selbst implementiert.

# 2 Homomorphe Verschlüsselung

## 2.1 Grundlagen moderner Verschlüsselung

Bevor die homomorphe Verschlüsselung dargestellt werden kann, stelle ich ein grundlegendes Verständnis her. Einleitend halte ich fest, dass die klassische Verschlüsselung für dieses Thema nicht relevant ist, da diese in modernen Systemen nicht verwendet wird. Aus diesem Grund werden nur moderne Verfahren angesprochen. Hier werden die beiden relevanten Arten der modernen Verschlüsselung vorgestellt.

Die modernen Verschlüsselungen werden in zwei Gruppen eingeteilt: Symmetrische Verschlüsselung (Abbildung 1: Aufbau einer symmetrischen Verschlüsselung) und asymmetrische Verschlüsslung (Abbildung 2: Aufbau einer asymmetrischen Verschlüsselung).[9]

### 2.1.1 Symmetrische Verschlüsselung

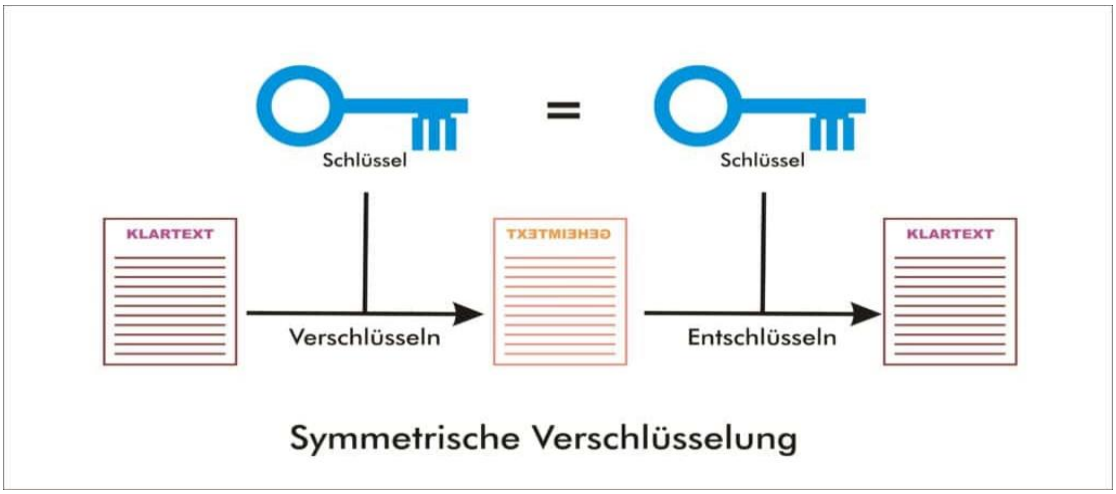


*Abbildung 1: Aufbau einer symmetrischen Verschlüsselung*[10]

Bei einer symmetrischen Verschlüsselung wird mit nur einem geheimen Schlüssel gearbeitet. Um eine Nachricht zu verschlüsseln, kann eine Verschlüsselungsfunktion *e* folgendermaßen

[9] Vgl. Hartmut Ernst / Jochen Schmidt / Gerd Beneken, »Verschlüsselung«, in: Hartmut Ernst / Jochen Schmidt / Gerd Beneken (Hrsg.), *Grundkurs Informatik, Grundlagen und Konzepte für die erfolgreiche IT-Praxis – eine umfassende, praxisorientierte Einführung,* Wiesbaden 2020, S. 137–172, hier: S. 138.

[10] Richard Meusers. *Welche Verschlüsselungsstandards gibt es in AV-Systemen?*, https://www.professional-system.de/basics/netzwerk-im-tarnmodus-verschluesselungsstandards-in-av-systemen/, Stand: 23.10.2022.

angewendet werden: „$y = e(x, k)$“[11]. Hier beschreibt $y$ den verschlüsselten Text, der aus dem, mit dem Schlüssel $k$ verschlüsselten, Ausgangstext $x$ entsteht.[12]

Damit aber aus den verschlüsselten Informationen, wieder eine korrekte Nachricht wird, wird eine Entschlüsselungsfunktion $d$ folgendermaßen verwendet: „$x = d(y, k)$“[13]. Hier ist das $x$ die ursprüngliche Nachricht, $y$ der verschlüsselte Text und $k$ der geheime Schlüssel.[14]

### 2.1.2 Asymmetrische Verschlüsselung

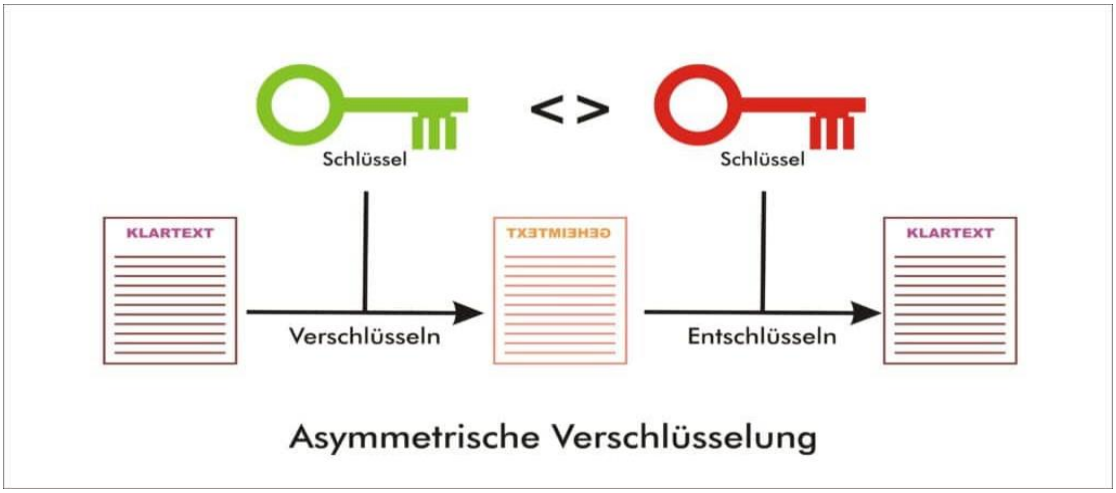


*Abbildung 2: Aufbau einer asymmetrischen Verschlüsselung*[15]

Der Unterschied einer asymmetrischen Verschlüsselung, welche auch Public-Key-Verschlüsselung genannt wird, zu einer symmetrischen Verschlüsselung besteht darin, dass bei einer Asymmetrischen Verschlüsselung zwei verschiedene Schlüssel verwendet werden. Einen öffentlichen Schlüssel, welcher zum Verschlüsseln verwendet wird, und einen privaten Schlüssel, der zum Entschlüsseln verwendet wird.[16] Dieses Verfahren, ist von Dr. Martin Hellman und Whitfield Diffi erfunden worden, als Ersatz für die Ver- und Entschlüsselung von Daten, mit nur einem Schlüssel.[17] Der genaue Unterschied zwischen diesen beiden Schlüsseln soll später thematisiert werden.

---

[11] Ernst / Schmidt / Beneken, Verschlüsselung, S. 139.
[12] Vgl. ebd.
[13] Ebd.
[14] Vgl. ebd.
[15] Richard Meusers, Welche Verschlüsselungsstandards gibt es in AV-Systemen?
[16] Vgl. R. L. Rivest / A. Shamir / L. Adleman, »A method for obtaining digital signatures and public-key cryptosystems«, *Commun. ACM*, 21, 1978, S. 120–126, hier: S. 120–121.
[17] Vgl. Gina Kolata, »SCIENTIST AT WORK: Leonard Adleman; Hitting the High Spots Of Computer Theory«, *New York Times*, 13, 1994.

Das Verschlüsseln kann man sich mit dieser Formel vorstellen: „$y = e(x, k_e)$“[18]. Die Verschlüsselungsfunktion $e$ verwendet nun den oben beschriebenen öffentlichen Schlüssel $k_e$ zum Verschlüsseln des Textes $x$.[19]

Um aus dem verschlüsselten Text den Originaltext zu dekodieren, wird eine Entschlüsselungsfunktion $d$ benötigt. Aber anstatt den verschlüsselten Text $y$ mit dem öffentlichen Schlüssel zu entschlüsseln, wird jetzt der private Schlüssel $k_d$ verwendet um wieder an den Ausgangstext $x$ zu kommen.[20] So erhält man die Formel: „$x = d(y, k_d)$“[21].

## 2.2 Aufbau homomorpher Verschlüsselung

Homomorphe Verschlüsslung soll ein Problem beseitigen, das die meisten anderen Verschlüsselungsarten aufweisen: Die Ausführung von Berechnungen auf verschlüsselten Daten. Das heißt, im konkreten Fall, wenn man Daten in der Cloud codiert analysieren will, benötigt man eine Verschlüsselung welche homomorph ist.[22]

Die homomorphe Verschlüsselung nutzt das Prinzip des Homomorphismus aus der Mathematik.[23] Homomorphismus kann mit folgendem Beispiel veranschaulicht werden: Es gibt zwei Gruppen $(G,\circ)$ und $(H,\star)$. Wenn die Abbildung $\phi$: $\mathrm{G} \rightarrow \mathrm{H}$ homomorph sein soll, muss für jedes $a, b \in G$ folgendes gelten[24]: $\phi(\mathrm{a} \circ \mathrm{b}) = \phi(\mathrm{a}) \star \phi(\mathrm{b})$

Um zumindest in der Theorie zu ermöglichen, dass jeder eine Nachricht verschlüsseln - aber nicht entschlüsseln kann, verwendet die homomorphe Verschlüsselung das asymmetrische bzw. das Public-Key Verschlüsselungsverfahren.[25]

Zusätzlich anzumerken ist, dass nicht nur arithmetische Operationen verschlüsselt durchgeführt werden können, sondern eine Unterkategorie dieser Verschlüsselung, namens voll-homomorphe Verschlüsselung, zudem die Verwendung von booleschen Schaltkreisen erlaubt.[26]

---

[18] Ernst / Schmidt / Beneken, Verschlüsselung, S. 139.
[19] Vgl. ebd.
[20] Vgl. ebd.
[21] Ebd.
[22] Vgl. Thomas Englert. *Evaluation des praktischen Nutzen von homomorpher Verschlüsselung* 10.09.2021, S. 5.
[23] Vgl. Niklas A. Zbick. *Homomorphe Verschlüsselung*, Dortmund 28. Juni 2012, S. 5.
[24] Vgl. Tobias Glosauer, »Homomorphismen«, in: Tobias Glosauer (Hrsg.), *Elementar(st)e Gruppentheorie,* Wiesbaden 2016, S. 69–79, hier: S. 69.
[25] Vgl. Englert, Evaluation des praktischen Nutzen von homomorpher Verschlüsselung, S. 6.
[26] Vgl. Adil Bouti. *Homomorphe Verschlüsselung und Cloud-Computing*, Hagen 2020, S. 49–50.

# 3 Arten homomorpher Verschlüsselung

Homomorphe Verschlüsselung ist der Überbegriff für diverse Verschlüsselungen, welche homomorph sind. Jede dieser Untergruppen dieser Verschlüsselungsart unterscheidet sich in Details, wie der Art der mathematischen Operationen und deren Häufigkeit der Ausführung. Deshalb wird Homomorphe Verschlüsselung in zwei Unterkategorien eingeteilt: Teil- und Voll-homomorphe Verschlüsselung.[27]

## 3.1 Teil-homomorphe Verschlüsselung

Die Beschränkung der mathematischen Operationen, wie oben aufgeführt, wird bei dieser Gruppe besonders deutlich. Das Hauptmerkmal dieser Unterkategorie liegt darin, dass ein verschlüsselter Text nur in einer Disziplin homomorph ist. Dies meint, dass nur eine bestimmte mathematische Operation verschlüsselt ausgeführt werden kann. Das wird auch homomorphe Eigenschaft genannt. Man unterscheidet hier oft zwischen multiplikativ homomorph und additiv homomorph.[28]

### 3.1.1 RSA

Eine bekannte Verschlüsselung welche multiplikativ homomorph ist, heißt RSA.[29] RSA ist ein Kürzel für die Anfangsbuchstaben der drei Erfinder Dr. Ronald Rivest, Dr. Adi Shamir und Dr. Leonard Adleman. Zudem verwendet RSA auch eine asymmetrische Verschlüsselung.[30] Bevor ein Bezug zum Homomorphismus hergestellt werden kann, muss sich erst das Wissen, wie die Schlüssel erstellt werden, wie ver- und entschlüsselt wird, vorhanden sein.

#### *3.1.1.1 Aufbau von RSA*

Die Ver- und das Entschlüsselungsverfahren sind nicht komplex. Die Verschlüsselungsfunktion nennen wird hier als $E(M)$ bezeichnet, bei der $M$ der zu verschlüsselnde Text ist. Diese Funktion berechnet $M$ hoch $e$ Modulo $N$. Das Ergebnis ist der Geheimtext, $C$. Die Entschlüsselungsfunktion, $D(C)$ berechnet $C$ hoch eine Zahl $d$ Modulo $n$. Daraus ergeben sich folgende Funktionen: „$\mathrm{C} \equiv \mathrm{E(M)} \equiv \mathrm{M^e (mod\ N)}$ *[...]* $\mathrm{D(C)} \equiv \mathrm{C^d (mod\ n)}$“[31].

---

[27] Vgl. Englert, Evaluation des praktischen Nutzen von homomorpher Verschlüsselung, S. 6.
[28] Vgl. Bouti, Homomorphe Verschlüsselung und Cloud-Computing, S. 51–52.
[29] Vgl. K. E. Staniewicz. *A Fully Homomorphic Encryption scheme* 2016, S. 3.
[30] Vgl. Kolata, SCIENTIST AT WORK: Leonard Adleman; Hitting the High Spots Of Computer Theory.
[31] Rivest / Shamir / Adleman, A method for obtaining digital signatures and public-key cryptosystems, S. 122.

Wie im Kapitel 2.1.2 oben dargestellt, benötigt eine asymmetrische Verschlüsselung einen öffentlichen und einen privaten Schlüssel. In der Codierung und Decodierung sind Variablen verwendet worden, die noch nicht definiert wurden: $e$, $d$ und $N$. Diese Variablen sind Teile der einzelnen Schlüssel. Der öffentliche Schlüssel besteht aus $e$ und $n$, der private Schlüssel aus $d$ und $n$.[32] Um anschaulicher darzustellen, wie die Schlüssel erzeugt werden, wird jeweils ein Beispiel für jeden Schritt angeben.

Der erste Schritt besteht daraus zwei Primzahlen $p$ und $q$ auszuwählen, welche eigentlich möglichst groß sein sollten; es hier aber zu Veranschaulichungszwecken nicht sind. In diesem Fall ($p = 11; q = 13$). Mit diesen zwei Primzahlen, kann man $n$ berechnen, bei dem es sich um einen Bestandteil der beiden Schlüssel handelt. $n$ ist das Produkt der beiden Primzahlen: $n = p * q$[33]. Hier im Beispiel gilt: $n = 11 * 13 = 143$. Im nächsten Schritt muss die Eulersche Phi-Funktion, berechnet werden. Sie ermittelt *„die Anzahl aller zu* n *teilerfremden Zahlen, die kleiner sind als* n.*“*[34].

Daraus ergibt sich: $\phi(n) = (p - 1) * (q - 1).$

In dem Beispiel würde dann folgendes gelten: $\phi(n) = (10 * 12) = 120$ .

Um den Prozess abzuschließen, muss nur noch der öffentliche Exponent $e$ und der private Exponent $d$ bestimmen werden. Sie müssen folgende Eigenschaft nachweisen: [35] „ $ed \equiv 1 (\mathrm{mod}\, \phi(n))$“[36].

Zuerst muss $e$ bestimmt werden. Dabei handelt es sich, wie oben beschrieben, um einen Teil des öffentlichen Schlüssels und somit um den Exponenten beim Verschlüsseln. Dieser muss so gewählt werden, dass er größer als null und kleiner als $n$ ist. Ferner ist es von Bedeutung, dass dieser teilerfremd zu der oben gezeigten $\phi(n)$ Funktion ist.[37] Eine Zahl ist zu einer anderen teilerfremd, wenn der größte gemeinsame Teiler 1 ist.[38]

---

[32] Vgl. ebd.
[33] Vgl. ebd., S. 123.
[34] Albrecht Beutelspacher / Heike B. Neumann / Thomas Schwarzpaul, »Der RSA-Algorithmus«, in: Albrecht Beutelspacher / Heike B. Neumann / Thomas Schwarzpaul (Hrsg.), *Kryptografie in Theorie und Praxis: Mathematische Grundlagen für Internetsicherheit, Mobilfunk und elektronisches Geld,* Wiesbaden 2010, S. 115–131, hier: S. 116.
[35] Vgl. ebd., S. 115–116.
[36] Rivest / Shamir / Adleman, A method for obtaining digital signatures and public-key cryptosystems, S. 123.
[37] Vgl. Beutelspacher / Neumann / Schwarzpaul, Der RSA-Algorithmus, S. 115.
[38] Vgl. James S. Eaton. *A treatise on arithmetic*, Boston 1872, S. 49.

Mathematisch ausgedrückt: $ggt(e, \phi(n) = 1$

Im gewählten Beispiel: $e = 7$

Der finale Teil der Schlüsselerzeugung ist das inverse multiplikative Element von $e \bmod \phi(n)$ zu berechnen. Denn dies ist der Exponent $d$. Das inverse Element ist die Zahl mit der $e$ multipliziert werden muss damit das Ergebnis eins ist. In dem Fall des Beispiels, ist das inverse Element $d = 103$.[39]

Zusammengefasst bekommt man jetzt folgende Werten der Beispiele, den öffentlichen Schlüssel $(e = 7; n = 143)$ und den privaten Schlüssel $(d = 103; n = 143)$.

#### *3.1.1.2 RSA und homomorphe Verschlüsselung*

Nachdem der Aufbau einer RSA-Verschlüsselung nun grob skizziert wurde, wird sich dem Thema homomorphe Verschlüsselung wieder zurückgewendet. Im Kapitel 3.13.1, wurde bereits beschreiben, dass RSA zu den teil-homomorphen Verschlüsslungen zählt. Nun muss aber die Frage beantwortet werden, inwiefern RSA multiplikativ homomorph ist.

Man hat die Verschlüsselungsfunktion $E(m) = m^e \pmod n$. Dadurch dass RSA multiplikativ homomorph ist, muss es möglich sein zwei Ausgangstexten $m_1$ und $m_2$ verschlüsselt miteinander zu multiplizieren $E(m_1) * E(m_2)$, und das gleiche Ergebnis zu bekommen, wie im Falle der unverschlüsselten Multiplikation dieser Nachrichten und der anschließenden Verschlüsselung des Ergebnisses: $E(m_1 * m_2)$. Daraus kann folgendes abgeleitet werden:

$$\begin{aligned} E(m_1 * m_2) &= (m_1 m_2)^e \bmod n \\ &= m_1^e m_2^e \bmod n \\ &= E(m_1) * E(m_2) \end{aligned}$$

## 3.2 Voll-homomorphe Verschlüsselung

Jedoch haben teil-homomorphe Verschlüsselungen in der Regel ein Problem: es ist nur möglich eine einzige mathematische Operation auszuführen. Daher besteht der Bedarf an einer Verschlüsselung, mit der man verschlüsselt mehrere Rechenoperationen, wie Multiplikation und Addition ausführen kann. Für diesen Fall gibt es eine weitere Art der homomorphe Verschlüsselung. Die voll-homomorphe Verschlüsselung (engl. fully

[39] Vgl. Beutelspacher / Neumann / Schwarzpaul, Der RSA-Algorithmus, S. 120.

homomorphic encryption). Dies wurde schon 1978 von Rivest, Adlemann und Dertouzos beschrieben.[40]

Die voll-homomorphe Verschlüsslung benötigt eine andere Form des Homomorphismus. Und zwar den Ringhomomorphismus. Denn es soll möglich sein, Additionen und Multiplikationen auf verschlüsselten Daten auszuführen. In Kapitel 2.2 haben die Gruppen jeweils nur eine Verknüpfung, was die Ausführung auf eine bestimmte Operation beschränkt. Da hier aber Additionen und Multiplikationen ausgeführt werden sollen, benötigt es eine Alternative. Voll-homomorphe Verschlüsselung basiert daher auf etwas namens Ringhomomorphismus[41].

Ein Ring ist eine Menge mit zwei Verknüpfungen. Nun eine Abbildung $\varphi: R \to R'$ mit den Ringen $(R, +_R, \cdot_R); (R', \oplus_{R'}, \otimes_{R'})$ wird als Ringhomomorphismus deklariert, wenn für alle $a, b \in R$ folgendes gilt [42]:

$$\varphi(a +_R b) = \varphi(a) \oplus_{R'} \varphi(b)$$[43]
$$\varphi(a \cdot_R b) = \varphi(a) \otimes_{R'} \varphi(b)$$

### 3.2.1 Somewhat Homomorphic Encryption

Die Entwicklung voll-homomorphe Systeme wies oft starke Probleme auf. Das Problem war, dass zwar voll-homomorphe Eigenschaften vorhanden waren aber die Häufigkeit wie oft mathematische Operationen ausgeführt werden können war oftmals eingeschränkt. Die Verschlüsselungen, welche voll-homomorph sind, aber dieses Problem aufweisen, nennt man daher auch beschränkt homomorph (engl. somewhat homomorphic).[44] *„Der Grund dafür ist, dass die Verfahren auf Berechnungen mit Fehlern beruhen.“*[45]

# 4 Implementation einer homomorphen Verschlüsselung

Es gibt viele homomorphe Verschlüsselungen - aber nur eine Handvoll Verschlüsselungen, welche auch voll-homomorphe Eigenschaften aufweisen. Eine dieser Verschlüsselungen

---

[40] Vgl. Ronald L. Rivest / L. Adleman / Michael L. Dertouzos / others, »On data banks and privacy homomorphisms«, *Foundations of secure computation*, 4, 1978, S. 169–180.
[41] Vgl. Bouti, Homomorphe Verschlüsselung und Cloud-Computing, S. 54.
[42] Vgl. Siegfried Bosch, »Ringe und Polynome«, in: Siegfried Bosch (Hrsg.), *Algebra,* Berlin/Heidelberg 2020, S. 31–108, hier: S. 47.
[43] Vgl. ebd.
[44] Vgl. Zbick, Homomorphe Verschlüsselung, S. 5.
[45] Ebd., S. 6.

möchte ich hier mit einer Implementation veranschaulichen. Diese Verschlüsselung wird als BFV-Verschlüsselung genannt. BFV steht für Brakerski/Fan-Vercauteren. Diese Verschlüsselung wurde von Junfeng Fan und Frederik Vercauteren entwickelt, basiert aber auf einem Konzept von Zvika Brakerski.[46] Meine Implementation basiert auch auf dieser Veröffentlichung, jedoch wurde die Implementation einfacher gehalten. Dennoch kann jeder Schritt an dem Paper nachvollzogen werden, sowie bewiesen werden. Der Schwerpunkt wurde bei mir aber auf den einzelnen Mechanismen der Verschlüsselung und der Rechenoperationen gelegt. Der Programmcode ist sowohl im Anhang als auch in einem Git-Repository[47] zu finden.

BFV basiert auf dem RLWE (ring learning with error) Problem. Diese Verschlüsselungsart, verwendet daher keine Zahlen, sondern Polynome. Genauer Polynome die in einem von zwei Ringen definiert werden müssen: Dem Geheimtextring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ und einem Klartextring $R_t = \mathbb{Z}_t[x]/(x^n + 1)$. Zudem ist $t, q \in \mathbb{Z}$ und $n \in \mathbb{Z}$. Die Variablen $t$ und $q$, sind die Koeffizienten der jeweiligen Polynome. Wiederum gibt $n$, den Grad der Polynome an. Um dies zu gewährleisten, dass der Grad der Polynome und die Größe der Koeffizienten, die jeweiligen Werte nicht überschreitet, muss bei jeder Polynom Rechnung, das Ergebnis Modulo $q$ beziehungsweise $t$ und Modulo $x^n + 1$ genommen werden.

Für BFV ist zudem wichtig zu wissen, dass viele Parameter aus sogenannten Zufallsverteilungen kommen. Man unterscheidet zwischen einer Schlüsselverteilung $R_2$, einer Rechteckverteilung von $R_q$ und einer Normal-, bzw. Gauß-Verteilung $\mathcal{X}$. Zwar wird nicht genauer auf die Erlangung der Werte der für die Verteilungen eingegangen, aber für die Parameter der Gauß-Verteilung, werden im „homomorphic encryption standard“[48], Werte genannt, welche ich auch verwendet habe.

Zudem müssen ein paar Zeichen definiert werden. $[*]_q$ bedeutet, dass alles Modulo $q$ genommen wird. $\lfloor * \rceil$ bedeutet, dass das Ergebnis auf eine Ganzzahl gerundet wird.

---

[46] Vgl. Junfeng Fan / Frederik Vercauteren, »Somewhat Practical Fully Homomorphic Encryption«, 2012.
[47] Vgl. Leon Adomaitis. *https://github.com/GunniBusch/WTO_Projekt*.
[48] Vgl. Martin Albrecht / Melissa Chase / Hao Chen / Jintai Ding / Shafi Goldwasser / Sergey Gorbunov / Shai Halevi / Jeffrey Hoffstein / Kim Laine / Kristin Lauter / Satya Lokam / Daniele Micciancio / Dustin Moody / Travis Morrison / Amit Sahai / Vinod Vaikuntanathan. *Homomorphic Encryption Security Standard*, Toronto, Canada November 2018.

## 4.1 Encoding und Decoding

Bevor die Ver- und Entschlüsselung angesprochen werden kann, muss erst einmal festgehalten, werden, dass wir wie oben erwähnt mit Polynomen arbeiten. Aus diesem Grund müssen wir eine Technik finden, mit welcher Zahlen in Polynome umgewandelt werden können. Encoding ist also nicht anderes als die Darstellung, zum Beispiel einer Zahl zu ändern. Es gibt offensichtlich mehrere Möglichkeiten, wie man ein Ausganstext in ein Polynom des Klartextringes umwandelt.

Eine wichtige Anmerkung ist noch, dass Polynome als Koeffizienten in einem Array dargestellt werden.

Ein Polynom aus dem Array: $[1,0,3]$ wäre $1 + 0 * x^1 + 3 * x^2 = 1 + 3 * x^2$.

Ich habe mich entschieden, aufgrund der Einfachheit, den Ausgangstext so zu encodieren, dass der Ausgangstext in einem Polynom, eine Konstante ist. Um aber die Polynome immer auf gleicher Länge zu lassen, werden am Ende so viele Nullen hinzugefügt, dass damit die Länge der Arrays immer gleich sind.

```
    def encode(self, n, b):
        """
        Encodiert eine Zahl.
        :param n: zu encodierende Zahl
        :param b: Basis
        :return: Koeffizienten der Basis b Repräsentation von n
        """

        return np.array([n] + [0] * (self.n - 1), dtype=np.int64) % self.t
```

*Abbildung 3: Encode Funktion*

Beim Decodieren, wird einfach die erste Stelle aus dem Array genommen, und dass ist dann das Ergebnis.

## 4.2 Erzeugen der Schlüssel

In der BFV-Verschlüsselung, werden insgesamt drei Schlüssel benötigt: einen privaten Schlüssel, den ich $sk$ nenne; einen öffentlichen Schlüssel, der $pk$ genannt wird und einen Evaluations Schlüssel namens $ek$.

```
    def generateSecretKey(self):
        """
        Erzeugt privaten Schlüssel
        :return: privaten Schlüssel
        """
        sk = self._genPoly()

        return sk
```

*Abbildung 4: Erzeugung des Privaten Schlüssels*

Der private Schlüssel, ist einfach gesagt, ein Polynom mit den Koeffizienten, aus $\{-1, 0, +1\}$.

```
    def generatePublicKey(self, sk):
        """
        Erzeugt öffentlichen Schlüssel
        :param sk: Geheimer Schlüssel
        :return: öffentliche Schlüssel
        """

        a = self._genUniPoly()
        e = self._genNormalPoly()
        # pk1= -1(pk2*sk+e)=(pk2*(-sk)-e)
        pk1 = self._rnd(self.add(self.mul(-a, sk), -e) % self.q)
        pk2 = a

        return pk1, pk2
```

*Abbildung 5: Erzeugung eines öffentlichen Schlüssels*

Um den öffentlichen Schlüssel, zu erzeugen, welcher genau genommen nicht nur ein einziger Schlüssel ist, sondern sich aus zwei Schlüsseln zusammensetzt, benötigt man zum einen den privaten Schlüssel. Weiterhin wird auch nach einem Polynom e, mit zufälligen Koeffizienten aus der Verteilung $\mathcal{X}$, verlangt. Dies wird benötigt, um den ersten Teil des öffentlichen Schlüssels zu generieren. Wie man aber erkennen kann, wird für den ersten Teil auch noch der zweite Teil des Schlüssels benötigt. Dieser ist in meinem Programm als $a$ definiert, und $a$ ist ein Polynom aus der Verteilung $R_q$. Das Ganze wird miteinander verrechnet. Dafür gibt es die Funktionen „mul“ und „add“. Diese berechnen die Multiplikation bzw. Addition zwischen zwei Polynomen. Zusätzlich wird das Ergebnis Modulo mit dem Ringmodul $(x^n + 1)$ genommen. Mathematisch wird das folgendermaßen berechnet:

$$pk_1 = [-pk_2 * sk + (-e)]_q$$
$$pk_2 = a$$

Der Relinearisierungs oder Evaluations-Schlüssel $ek$ wird für die Multiplikation benötigt. Darauf komme ich später zurück.

```
    def generateRelinKey(self, sk):

        relin_key2 = self._genUniPoly(self.p * self.q)
        e = self._genNormalPoly()
        sk2 = P.polymul(sk, sk)
        secret_part = self.p * sk2

        relin_key1 = (self.add(self.add(
            self.mul(-relin_key2, sk),
            -e), secret_part)) % (self.p * self.q)

        return relin_key1, relin_key2
```

*Abbildung 6: Erzeugung des Evaluations Schlüssels*

Dieser Schlüssel ist genau wie der öffentliche Schlüssel in zwei Teile geteilt. Zusätzlich benötigen wir aber noch eine ganze Zahl $p$. Diese muss größer sein als $q$. Diese Zahl dient dazu, Rauschen zu minimieren, der bei der Multiplikation von Schlüsseltexten entsteht. Des Weiteren benötigt man den geheimen Schlüssel, der quadriert werden muss: $sk^2$. Dieser wird daraufhin mit der Konstante $p$ multipliziert. In meinem Code ist das die Variable $secret\_part$. Das weitere Verfahren ähnelt dem des öffentlichen Schlüssels. Nur muss bei dem ersten Teil, zusätzlich der secret_part addiert werden, sowie statt Modulo $q$, muss Modulo $p * q$ gerechnet werden.

$$ek_1 = [(-ek_2 * sk - e) + p * sk^2]_{p*q}$$
$$ek_2 = a$$

## 4.3 Verschlüsseln

Meine Implantation der BFV-Verschlüsslung verwendet zwei Klassen, die Klasse BFV und die Klasse CT. Die Verschlüsselung benötigt beide Klassen. Zwar befindet sich die Funktion zum Verschlüsseln in der Klasse BFV, aber die Funktion gibt als Schlüsseltext ein Objekt der Klasse CT zurück. Dieses Objekt hat die Schlüsseltext Polynome als Attribut gespeichert.

```
    def encrypt(self, pt, pk: tuple[np.ndarray, np.ndarray], ek:
tuple[np.ndarray, np.ndarray]) -> CT:
        """
        :param ek: evaluationKey. Gleich wie Relinearisierungsschlüssel
        :param pt: Codierter Ausgangstext
        :param pk: tuple des öffentlichen SSchlüssels
        :return: tuple der Verschlüsselten texte
        """
        pk1, pk2 = pk
        pt = self.encode(pt, self.encode_base)

        u = self._genPoly()
        e1 = self._genNormalPoly()
        e2 = self._genNormalPoly()
        quotient = (self.q // self.t)

        pk1u = (self.mul(pk1, u) % self.q)
        pk2u = (self.mul(pk2, u) % self.q)

        ct1 = self._rnd(self.add(pk1u, self.add(e1, pt * quotient)) %
self.q)
        ct2 = self._rnd(self.add(pk2u, e2) % self.q)

        return CT(self, (ct1, ct2), pk, ek)
```

*Abbildung 7: Verschlüsselungsfunktion*

Der erste Teil der Funktion sorgt dafür, dass ein übergebenes Integer zu einem Polynom wird (Encodieren). Dies wird dann als $pt$ bezeichnet. Dies wurde am Anfang angesprochen. Ähnlich wie bei den Schlüsseln, besteht der Chiffretext, aus zwei Teilen. Des Weiteren wird zum Verschlüsseln der öffentliche Schlüssel benötigt. Obwohl an die Funktion ein Relinearisierungs Schlüssel übergeben wird, wird dieser für die Verschlüsselung nicht benötigt, sondern zum Erzeugen des CT-Objektes am Ende. Weiterhin müssen zwei Polynome, $e_1, e_2$ aus $\mathcal{X}$, sowie ein Polynom $u$ aus der Rechteckverteilung $R_q$ erzeugt werden. Ein wichtiger Schritt für die Verschlüsselung ist, dass der Ausgangstext, um $\left\lfloor \frac{q}{t} \right\rfloor$ vergrößert wird. Mit diesen Parametern, entsteht folgendermaßen ein Schlüsseltextpaar:

$$C_1 = \left[ pk_1 * u + e_1 + \left\lfloor \frac{q}{t} \right\rfloor * pt \right]_q$$
$$C_2 = [pk_2 * u + e_2]_q$$

## 4.4 Entschlüsselung

```
    def decrypt(self, ct, sk):
        ct1, ct2 = ct.ct
        step1 = self._rnd((self.add(
            self.mul(ct2, sk), ct1
        ) % self.q))
        # Codierter Klartext
        dec = np.round(step1 / (self.q // self.t)) % self.t

        return np.int64(dec[0])
```

*Abbildung 8: Entschlüsselung*

Um einen Chiffretext zu entschlüsseln, wird zusätzlich zu dem Chiffretext, der private Schlüssel benötigt. Eine Besonderheit ist, dass am Ende nicht Modulo $q$ gerechnet wird, sondern Modulo $t$. Dies kommt daher, dass der Ausgangstext im Ring $R_t$ ist. Zusätzlich wird vorher das Ergebnis gerundet, um Rauschen zu entfernen. Die Entschlüsselung ist somit:

$$pt = \left[\left[\frac{t * [C_1 + C_2 * sk]_q}{q}\right]\right]_t$$

Zwar haben wir $pt$, aber das ist noch ein Polynom. Um einen Integer zu bekommen, muss dies erst decodiert werden. In diesem Fall ist einfach der Koeffizient mit dem niedrigsten Grad der gesuchte Wert.

## 4.5 Addition

Bei der Addition unterscheiden wir zwischen zwei Arten. Addition zwischen zwei Schlüsseltexten und zwischen einem Schlüsseltext und einem Integer, also einer ganzen Zahl.

```
    def __add__(self, other):

        ct11, ct12 = self.ct
        if isinstance(other, CT):
            ct21, ct22 = other.ct
            ct1_new = self._rnd(self.bfv.add(ct11, ct21) % self.bfv.q)
            ct2_new = self._rnd(self.bfv.add(ct12, ct22) % self.bfv.q)

            return CT(self.bfv, (ct1_new, ct2_new), self.pk, self.ek)
```

*Abbildung 9: Addition Teil 1*

Zuerst will ich klären, warum ich für die Schlüsseltexte eine eigene Klasse erstellt habe. Und zwar hat das, etwas mit der Addition, aber auch der Multiplikation zu tun. Die Funktion trägt den Namen $\_\_add\_\_$. Dieser ist nicht zufällig so gewählt worden. Wenn man in Python beispielsweise $3 + 2$ rechnet, dann wird die Funktion $\_\_add\_\_$ von der $3$ ausgeführt. Funktionieren würde auch, $(3).\_\_add\_\_(2)$. Nun da die Funktion überschrieben wurde, ist es

möglich, ein Objekt der Klasse $CT$ mit einem anderen Objekt zu addieren, indem ich das Pluszeichen verwende. Dies kann auch für andere Operatoren gemacht werden.

Die Addition zwischen zwei Schlüsseltexten, ist einfach die Addition der beiden Schlüsseltextpolynome. Bei der Addition mit einer ganzen Zahl, wird die Zahl vorher codiert, und dann verschlüsselt, ohne die Fehlerfunktionen zu verwenden.

## 4.6 Multiplikation

Die Multiplikation kann auch wieder so unterteilt werden wie bei der Addition. Die Multiplikation mit Ausganstexten ist ähnlich wie bei der Addition. Diesmal wird die Zahl aber nur codiert, und dann mit jeweils mit den beiden Teilen des Chiffrates multipliziert.

```
    def __mul__(self, other):

        ct11, ct12 = self.ct
        if isinstance(other, CT):
            ct21, ct22 = other.ct
            ct1 = np.int64(
                np.round(
                    self.bfv.mul(ct11, ct21) * self.bfv.t / self.bfv.q
                )) % self.bfv.q

            ct2 = np.int64(
                np.round(
                    self.bfv.add(self.bfv.mul(ct11, ct22),
                                 self.bfv.mul(ct12, ct21)
                                 ) * self.bfv.t / self.bfv.q
                )) % self.bfv.q

            ct3 = np.int64(
                np.round(
                    self.bfv.mul(ct12, ct22) * self.bfv.t / self.bfv.q)
            ) % self.bfv.q

            return CT(self.bfv, self._relin((ct1, ct2, ct3)), self.pk,
self.ek)
```

*Abbildung 10: Multiplikation von zwei Schlüsseltexten*

Die Multiplikation zweier Schlüsseltexte ist nicht trivial. Um das Ergebnis zu bekommen, muss man das Tensor Produkt berechnen. Und mit $\frac{t}{q}$ multipliziren. Der Haken ist, dass man dann drei statt zwei Schlüsseltexte bekommt:

$$C_1 = \left[\left[\left(\frac{t(ct_1^1 * ct_1^2)}{q}\right)\right]\right]_q$$
$$C_2 = \left[\left[\left(\frac{t(ct_1^1 * ct_1^2 + ct_2^1 * ct_1^2)}{q}\right)\right]\right]_q$$
$$c_3 = \left[\left[\left(\frac{t(ct_2^1 * ct_2^2)}{q}\right)\right]\right]_q$$

Um nun wieder auf zwei Schlüsseltexte zu kommen, benötigen wir etwas namens Relinearisierung.

## 4.7 Relinearisierung

Um die Anzahl der Chiffrate nach der Multiplikation von drei auf zwei zu reduzieren, benötigen wir die Relinearisierung:

```
    def _relin(self, ct: tuple[np.ndarray, np.ndarray, np.ndarray]) ->
tuple[np.ndarray, np.ndarray]:
        ct1, ct2, ct3 = ct
        ek1, ek2 = self.ek
        ct21 = np.int64(np.round(self.bfv.mul(ek1, ct3) / self.bfv.p)) %
self.bfv.q
        ct22 = np.int64(np.round(self.bfv.mul(ek2, ct3) / self.bfv.p)) %
self.bfv.q

        ct1 = np.int64(self.bfv.add(ct1, ct21)) % self.bfv.q
        ct2 = np.int64(self.bfv.add(ct2, ct22)) % self.bfv.q

        return ct1, ct2
```

*Abbildung 11: Relinearisierung*

Für diesen Prozess wird der Evaluations Schlüssel benötigt. Die Relinearisierung wird in zwei Schritte unterteilt.

Schritt 1:

$$C_1^* = \left[\left[\left(\frac{(ct_2 * ek_1)}{p}\right)\right]\right]_q$$
$$C_2^* = \left[\left[\left(\frac{(ct_2 * ek_2)}{p}\right)\right]\right]_q$$

Schritt 2:

$$nct_1 = [ct_1 + ct_2^*]_q$$
$$nct_2 = [ct_2 + c_2^*]_q$$

Die aus Schritt 2 gewonnen Variablen, können nun ausgegeben werden.

# 5 Entwicklungsstand

Die Bekanntheit homomorpher Verschlüsselung ist in den letzten Jahren deutlich gestiegen. Dies hat dazu geführt das Unternehmen wie Google[49] oder Microsoft[50] beginnen in Richtung homomorphe Verschlüsselung zu entwickeln. Denn das Problem der homomorphen Verschlüsslung liegt aktuell im Bereich der Geschwindigkeit. Um die Ausführung effizienter zu gestalten, entwickeln diese, sowie weitere Firmen, effizientere Möglichkeiten homomorphe, im speziellen voll homomorphe Verschlüsselungen umzusetzen. Obwohl teil-homomorphe Verschlüsselungen wie RSA bereits lange existieren, wurde die erste voll-homomorphe Verschlüsselung erst 2009 von Craig Gentry vorgestellt. Daher wird es vermutlich noch ein wenig Zeit in Anspruch nehmen, bis diese Lösungen marktreif sind. Die ersten Anwendungen könnten in der Medizin sein.

# 6 Fazit

Diese neue Art der Verschlüsselung wird voraussichtlich ein Meilenstein in der Geschichte des Datenschutzes darstellen. Insbesondere da Datenschutz und Cloud Computing nun miteinander vereinbar sind, sehe ich viel Potenzial in dieser Technologie. Zwar ist diese Technologie noch nicht in Verwendung, könnte aber ganze Sektoren, wie das Gesundheitswesen verändern, da eine verschlüsselte Auswertung von Patientendaten in der Public Cloud möglich werden könnte – vorbehaltlich gesetzlicher Regulierung. Dies war bislang aufgrund Beschränklungen des Datenschutzes nicht möglich.

---

[49] Vgl. Shruthi Gorantala / Rob Springer / Sean Purser-Haskell / William Lam / Royce Wilson / Asra Ali / Eric P. Astor / Itai Zukerman / Sam Ruth / Christoph Dibak / Phillipp Schoppmann / Sasha Kulankhina / Alain Forget / David Marn / Cameron Tew / Rafael Misoczki / Bernat Guillen / Xinyu Ye / Dennis Kraft / Damien Desfontaines / Aishe Krishnamurthy / Miguel Guevara / Irippuge Milinda Perera / Yurii Sushko / Bryant Gipson, »A General Purpose Transpiler for Fully Homomorphic Encryption«, 2021.

[50] Vgl. Microsoft, »Microsoft SEAL (release 4.0)«, 2022.

# 7 Literaturverzeichnis

Adomaitis, Leon: https://github.com/GunniBusch/WTO_Projekt.

Albrecht, Martin/Chase, Melissa/Chen, Hao/Ding, Jintai/Goldwasser, Shafi/Gorbunov, Sergey/Halevi, Shai/Hoffstein, Jeffrey/Laine, Kim/Lauter, Kristin/Lokam, Satya/Micciancio, Daniele/Moody, Dustin/Morrison, Travis/Sahai, Amit/Vaikuntanathan, Vinod: Homomorphic Encryption Security Standard. Toronto, Canada November 2018.

Beutelspacher, Albrecht/Neumann, Heike B./Schwarzpaul, Thomas: Der RSA-Algorithmus. In: Albrecht Beutelspacher / Heike B. Neumann / Thomas Schwarzpaul (Hrsg.): Kryptografie in Theorie und Praxis: Mathematische Grundlagen für Internetsicherheit, Mobilfunk und elektronisches Geld. Wiesbaden 2010. S. 115–131.

Bosch, Siegfried: Ringe und Polynome. In: Siegfried Bosch (Hrsg.): Algebra. Berlin/Heidelberg [9]2020. S. 31–108.

Bouti, Adil: Homomorphe Verschlüsselung und Cloud-Computing. Hagen 2020.

Eaton, James S.: A treatise on arithmetic. Boston 1872.

Englert, Thomas: Evaluation des praktischen Nutzen von homomorpher Verschlüsselung 10.09.2021.

Ernst, Hartmut/Schmidt, Jochen/Beneken, Gerd: Verschlüsselung. In: Hartmut Ernst / Jochen Schmidt / Gerd Beneken (Hrsg.): Grundkurs Informatik. Grundlagen und Konzepte für die erfolgreiche IT-Praxis – eine umfassende, praxisorientierte Einführung. Wiesbaden [7]2020. S. 137–172.

Fan, Junfeng/Vercauteren, Frederik: Somewhat Practical Fully Homomorphic Encryption (2012).

Glosauer, Tobias: Homomorphismen. In: Tobias Glosauer (Hrsg.): Elementar(st)e Gruppentheorie. Wiesbaden 2016. S. 69–79.

Gorantala, Shruthi/Springer, Rob/Purser-Haskell, Sean/Lam, William/Wilson, Royce/Ali, Asra/Astor, Eric P./Zukerman, Itai/Ruth, Sam/Dibak, Christoph/Schoppmann, Phillipp/Kulankhina, Sasha/Forget, Alain/Marn, David/Tew, Cameron/Misoczki, Rafael/Guillen, Bernat/Ye, Xinyu/Kraft, Dennis/Desfontaines, Damien/Krishnamurthy, Aishe/Guevara, Miguel/Perera, Irippuge Milinda/Sushko, Yurii/Gipson, Bryant: A General Purpose Transpiler for Fully Homomorphic Encryption (2021).

Kolata, Gina: SCIENTIST AT WORK: Leonard Adleman; Hitting the High Spots Of Computer Theory. In: New York Times 13 (1994).

Microsoft: Microsoft SEAL (release 4.0) (2022).

Richard Meusers: Welche Verschlüsselungsstandards gibt es in AV-Systemen? https://www.professional-system.de/basics/netzwerk-im-tarnmodus-verschluesselungsstandards-in-av-systemen/. Stand: 23.10.2022.

Rivest, R. L.; Shamir, A.; Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. In: Communications of the ACM 21 (1978) S. 120–126.

Rivest, Ronald L.; Adleman, L.; Dertouzos, Michael L.; others: On data banks and privacy homomorphisms. In: Foundations of secure computation 4 (1978) S. 169–180.

Staniewicz, K. E.: A Fully Homomorphic Encryption scheme 2016.

Vogel, Robert/Koçoğlu, Tarkan/Berger, Thomas: Cloud Computing. In: Robert Vogel / Tarkan Koçoğlu / Thomas Berger (Hrsg.): Desktopvirtualisierung: Definitionen – Architekturen – Business-Nutzen Mit 35 Abbildungen und 16 Tabellen. Wiesbaden 2010. S. 119–137.

Weichert, Thilo: Cloud Computing und Datenschutz. In: Datenschutz und Datensicherheit - DuD 34 (2010) S. 679–687.

Wiese, Lena/Homann, Daniel/Waage, Tim/Brenner, Michael: Homomorphe Verschlüsselung für Cloud-Datenbanken: Übersicht und Anforderungsanalyse. In: Hanno Langweg / Michael Meier / Bernhard C. Witt / Delphine Reinhardt (Hrsg.): SICHERHEIT 2018. Bonn 2018. S. 221–234.

Zbick, Niklas A.: Homomorphe Verschlüsselung. Dortmund 28. Juni 2012.

# 8 Abbildungsverzeichnis

# 9 Anhang

Der gesamte Code zur Implementation kann auf GitHub gefunden werden. Der Link dazu ist: https://github.com/GunniBusch/WTO_Projekt. Der Commit unter dem diese Version gefunden werden kann, ist: a83b3fcda61745d0ce0c4ad0ce9c1bf02e5c7354. Außerdem zu finden unter dem Tag: W-Sem.

## 9.1 Implementation der Klasse BFV

```
1.  import numpy as np
2.  from numpy.polynomial import polynomial as P
3.
4.  from BFV import CT
5.
6.
7.  class BFV:
8.      def __init__(self, ring_size: int, koef_pt: int, koef_ct: int, poly_modulo:
    np.ndarray,
9.                   modulo_relin: np.ndarray, mean, scale, encode_base):
10.         """
11.
12.         :param ring_size: Größe des Polynomrings
13.         :param koef_pt: Koeffizienten des Ausgangstextes
14.         :param koef_ct: Koeffizienten des Chiffretextes
15.         :param poly_modulo: Modulo Polynom
16.         :param mean: Median
17.         :param scale: Standardabweichung
18.         """
19.         # Ring größe
20.         self.n = ring_size
21.         # Chiffretext polynom Koeffizienten
22.         self.q = koef_ct
23.         # Ausgangstext polynom koeffizienten
24.         self.t = koef_pt
25.
26.         # (x^n+1)
27.         self.poly_modulo = poly_modulo
28.         self.p = modulo_relin
29.         self.mean = mean
30.         self.scale = scale
31.         self.encode_base = encode_base
32.
33.     # Polynom Rechnungen
34.     @staticmethod
35.     def _rnd(x):
36.
37.         return np.int64(np.round(x))
38.
39.     def add(self, poly1: np.ndarray, poly2: np.ndarray) -> np.ndarray:
40.
41.         xy = P.polyadd(poly1, poly2)
42.
43.         return P.polydiv(xy, self.poly_modulo)[1]
44.
45.     def mul(self, poly1: np.ndarray, poly2=None) -> np.ndarray:
46.
47.         if poly2 is None:
48.             poly2 = poly1
49.
50.         xy = P.polymul(poly1, poly2)
51.
```

```
52.         return P.polydiv(xy, self.poly_modulo)[1]
53.
54.     def _genPoly(self):
55.         """
56.
57.         :return: Koeffizienten eines Polynoms des Grades n-1
58.         """
59.         rng = np.random.default_rng()
60.         return rng.integers(low=-1, high=2, size=self.n)
61.
62.     def _genNormalPoly(self):
63.         """
64.
65.         :return: Koeffizienten in einer Normalverteilung eines Polynoms des Grades n-1
66.         """
67.         rng = np.random.default_rng()
68.         return np.int64(rng.normal(self.mean, self.scale, self.n))
69.
70.     def _genUniPoly(self, mod=None):
71.         """
72.
73.         :return: Koeffizienten in Z_n eines Polynoms des Grades n-1
74.         """
75.
76.         if mod is None:
77.             mod = self.q
78.
79.         rng = np.random.default_rng(np.random.PCG64DXSM())
80.         return rng.integers(low=0, high=mod, size=self.n)
81.
82.     def generateSecretKey(self):
83.         """
84.         Erzeugt privaten Schlüssel
85.         :return: privaten Schlüssel und öffentliche Schlüssel (sk,(pk1,pk2))
86.         """
87.         sk = self._genPoly()
88.
89.         return sk
90.
91.     def generatePublicKey(self, sk):
92.         """
93.         Erzeugt öffentlichen Schlüssel
94.         :param sk: Geheimer Schlüssel
95.         :return: privaten Schlüssel und öffentliche Schlüssel (sk,(pk1,pk2))
96.         """
97.
98.         a = self._genUniPoly()
99.         e = self._genNormalPoly()
100.             # pk1= -1(pk2*sk+e)=(pk2*(-sk)-e)
101.             pk1 = self._rnd(self.add(self.mul(-a, sk), -e) % self.q)
102.             pk2 = a
103.
104.             return pk1, pk2
105.
106.         def generateRelinKey(self, sk):
107.
108.             relin_key2 = self._genUniPoly(self.p * self.q)
109.             e = self._genNormalPoly()
110.             sk2 = P.polymul(sk, sk)
111.             secret_part = self.p * sk2
112.
113.             relin_key1 = (self.add(self.add(
114.                 self.mul(-relin_key2, sk),
115.                 -e), secret_part)) % (self.p * self.q)
116.
117.             return relin_key1, relin_key2
118.
```

```
119.          def encode(self, n, b):
120.              """
121.              Encodiert eine Zahl.
122.              :param n: zu encodierende Zahl
123.              :param b: Basis
124.              :return: Koeffizienten der Basis b Repräsentation von n
125.              """
126.
127.              return np.array([n] + [0] * (self.n - 1), dtype=np.int64) % self.t
128.
129.          @staticmethod
130.          def decoden, b):
131.              """
132.              Encodiert eine Zahl.
133.              :param n: zu encodierende Zahl
134.              :param b: Basis
135.              :return: Koeffizienten der Basis b Repräsentation von n
136.              """
137.              return P.polyval(b, n)
138.
139.          def   encrypt(self,   pt,   pk:   tuple[np.ndarray,   np.ndarray],   ek:
    tuple[np.ndarray, np.ndarray]) -> CT:
140.              """
141.
142.              :param ek: evaluationKey. Gleich wie Relinearisierungsschlüssel
143.              :param pt: Codierter Ausgangstext
144.              :param pk: tuple des öffentlichen SSchlüssels
145.              :return: tuple der Verschlüsselten texte
146.              """
147.              pk1, pk2 = pk
148.              pt = self.encode(pt, self.encode_base)
149.
150.              u = self._genPoly()
151.              e1 = self._genNormalPoly()
152.              e2 = self._genNormalPoly()
153.              quotient = (self.q // self.t)
154.
155.              pk1u = (self.mul(pk1, u) % self.q)
156.              pk2u = (self.mul(pk2, u) % self.q)
157.
158.              ct1 = self._rnd(self.add(pk1u, self.add(e1, pt * quotient)) % self.q)
159.              ct2 = self._rnd(self.add(pk2u, e2) % self.q)
160.
161.              return CT(self, (ct1, ct2), pk, ek)
162.
163.          def decrypt(self, ct, sk):
164.              ct1, ct2 = ct.ct
165.              step1 = self._rnd((self.add(
166.                  self.mul(ct2, sk), ct1
167.              ) % self.q))
168.              # Codierter Klartext
169.              dec = np.round(step1 / (self.q // self.t)) % self.t
170.
171.              return np.int64(dec[0])
```

## 9.2 Implementation der Klasse CT

```
1.  import numpy as np
2.
3.  import BFV
4.
5.
6.  class CT:
7.      def __init__(self, bfv: BFV, ct: tuple[np.ndarray, np.ndarray], pk, ek):
8.
```

```
9.          self.bfv = bfv
10.         self.ct = ct
11.         self.pk = pk
12.         self.ek = ek
13.
14.     def   _relin(self,   ct:   tuple[np.ndarray,   np.ndarray,   np.ndarray])   ->
    tuple[np.ndarray, np.ndarray]:
15.         ct1, ct2, ct3 = ct
16.         ek1, ek2 = self.ek
17.         ct21 = np.int64(np.round(self.bfv.mul(ek1, ct3) / self.bfv.p)) % self.bfv.q
18.         ct22 = np.int64(np.round(self.bfv.mul(ek2, ct3) / self.bfv.p)) % self.bfv.q
19.
20.         ct1 = np.int64(self.bfv.add(ct1, ct21)) % self.bfv.q
21.         ct2 = np.int64(self.bfv.add(ct2, ct22)) % self.bfv.q
22.
23.         return ct1, ct2
24.
25.     def _rnd(self, x):
26.         return self.bfv._rnd(x)
27.
28.     def __add__(self, other):
29.
30.         ct11, ct12 = self.ct
31.         if isinstance(other, CT):
32.             ct21, ct22 = other.ct
33.             ct1_new = self._rnd(self.bfv.add(ct11, ct21) % self.bfv.q)
34.             ct2_new = self._rnd(self.bfv.add(ct12, ct22) % self.bfv.q)
35.
36.             return CT(self.bfv, (ct1_new, ct2_new), self.pk, self.ek)
37.
38.         elif isinstance(other, int):
39.
40.             ct1, ct2 = self.ct
41.
42.             size = self.bfv.n - 1
43.
44.             m = np.array([other] + [0] * (size), dtype=np.int64) % self.bfv.t
45.
46.             d = self.bfv.q // self.bfv.t
47.
48.             scaled_m = d * m
49.
50.             new_ct1 = self.bfv.add(ct1, scaled_m) % self.bfv.q
51.
52.             return CT(self.bfv, (new_ct1, ct2), self.pk, self.ek)
53.         else:
54.             raise ValueError(f"can only add ciphertexts or integers, but operand was
    {other.__class__}")
55.
56.     def __mul__(self, other):
57.
58.         ct11, ct12 = self.ct
59.         if isinstance(other, CT):
60.             ct21, ct22 = other.ct
61.             ct1 = np.int64(
62.                 np.round(
63.                     self.bfv.mul(ct11, ct21) * self.bfv.t / self.bfv.q
64.                 )) % self.bfv.q
65.
66.             ct2 = np.int64(
67.                 np.round(
68.                     self.bfv.add(self.bfv.mul(ct11, ct22),
69.                                  self.bfv.mul(ct12, ct21)
70.                                  ) * self.bfv.t / self.bfv.q
71.                 )) % self.bfv.q
72.
73.             ct3 = np.int64(
```

```
74.                 np.round(
75.                     self.bfv.mul(ct12, ct22) * self.bfv.t / self.bfv.q)
76.             ) % self.bfv.q
77.
78.             return CT(self.bfv, self._relin((ct1, ct2, ct3)), self.pk, self.ek)
79.
80.         elif isinstance(other, int):
81.
82.             ct1, ct2 = self.ct
83.             size = len(self.bfv.poly_modulo)
84.
85.             m = np.array([other] + [0] * (size - 2), dtype=np.int64) % self.bfv.t
86.             ct_new_1 = self.bfv.mul(ct1, m) % self.bfv.q
87.             ct_new_2 = self.bfv.mul(ct2, m) % self.bfv.q
88.
89.             return CT(self.bfv, (ct_new_1, ct_new_2), self.pk, self.ek)
90.         else:
91.             raise ValueError(f"can only multiply with ciphertexts or integers, but
    operand was {other.__class__}")
```