Hochschule für Technik und Wirtschaft Dresden

Bachelorarbeit

Implementierung eines Post-Quanten-Kryptografischen Algorithmus in Ada/SPARK und Verifikation einiger Konzepte

Marinovic, Tom

Betreuer:
Prof. Dr. Andreas Westfeld
Zweit-Gutachter:
Prof. Dr. Robert Baumgartl

Bearbeitet vom:
07.08.2020 bis zum 02.10.2020

September 28, 2020

Inhaltsverzeichnis

Abkürzungen

BSI Bundesamt für Sicherheit in der Informationstechnik. 6, 16

IEEE Institute of Electrical and Electronics Engineers. 17

NISQ Noisy Intermediate-Scale Quantum. 6, 10

NIST National Institute of Standards and Technology. 16, 31

PQC Post-Quanten-Kryptografie. 6, 8, 33, 34

QC Quanten-Computer. 6, 7, 9, 10, 13, 34

SQUID Superconducting quantum interference device. 10

SVP Shortest Vector Problem. 18, 19

Glossar

Blow-UP-Faktor Beschreibt wie sehr sich die Größe einer verschlüsselten Datei von ihrer Ausgangsgröße unterscheidet. 6

Brute-Force Ein Angriff auf ein verschlüsseltes System, bei welchem mögliche Passwörter durchprobiert werden, ohne einen Anhaltspunkt zu haben. 10

Chosen-Plaintext-Attack Ein Angriffsmodel auf ein Kryptosystem, bei welchem der Angreifer Zugriff auf das Chiffrat für einen beliebigen Klarttext hat. 18

Mooresches Gesetz Eine Regel nach Gordon Moore, welcher entsprechend sich die Komplexität integrierter Schaltkreise regelmäßgig verdoppelt. Seid 1971 bestehend, konnte dieses Gesetz noch nicht widerlegt werden.. 11

NP-Vollständig Bezeichnet ein Problem in der Informatik, welches sich nichtdeterministisch in Polynominalzeit lösen lässt. In anderen Worten sind es Probleme deren Rechenaufwand in Polynominaler Zeit steigt. 12, 13, 16–18

Qubit Auch QBit; Zweizustands Quantensystem, welches im Quanten Computer analog zum klassischen Bit dient. 9

Seed Beschreibt einen Wert, welcher einem Generator übergeben wird zur berechnung, ein gleicher Seed führt zu gleichem Ergebnis. 22

Thread Ein geschlossener Teil eines Prozesses zur Ausführung eines Programmabschnitts. 28

Trap-Door-Funktion Eine Funktion, für welche das finden der Inverse ohne Zusatzinformationen deutlich aufwendiger ist, als die Ausführung der Funktion selbst (in Rechenzeit). 27

Zero-Knowledge Ein Angreifer sollte keinerlei Informationen, abgesehen von Gültigkeit der Ausführung, über ein System aus dessen Arbeitsweise heraus erhalten können. 27

Abbildungsverzeichnis

1 Einführung

1.1 Themen Beschreibung

Ziel dieser Arbeit ist die Implementierung eines Post-Quanten sicheren Algorithmus in der Programmiersprache Ada mit dem Subset SPARK. Die Eigenheiten dieser Sprache werden in ihrem eigenen Gliederungspunkt (2.4) näher beleuchtet. Um eine fundierte Entscheidung über die Auswahl zu treffen, müssen Post-Quanten-Kryptografische Algorithmen verglichen werden. Die Wahl der Kandidaten sollte verschiedene Faktoren berücksichtigen, wie Komplexität, Quantensicherheit, Dokumentation und Verbreitung.
Anschließend sollen mehrere Konzepte an diesem Algorithmus analysiert werden, unter anderem das Verhalten der Schlüsselgrößen und des Blow-UP-Faktors. Mittels SPARK soll der Algorithmus verifiziert und einzelne Konzepte exemplarisch bewiesen werden. Die verschiedenen Subkonzepte von SPARK sollten gezeigt werden. Abschließend wird der Algorithmus aufgrund seiner Eigenschaften bewertet und ein Fazit zur Eignung im praktischen Einsatz gezogen.

1.2 Hintergrund

Diese Bachelorarbeit beschäftigt sich mit Post-Quanten-Kryptografie, folgend PQC. Dieses Teilgebiet fokussiert sich auf das Szenario eines existierenden Quantencomputers, welcher in der Lage ist Quantenalgorithmen zuverlässig auszuführen. Ziel ist es Kryptografische Systeme zu entwickeln, welche gegenüber QC und klassischen Computern sicher sind. Heutige Quanten-Computer gehören zu der Kategorie Noisy Intermediate-Scale Quantum Computer, welche eine zu hohe Fehlerrate besitzt um effektiv beliebige Quantenalgorithmen auszuführen. Sie müssen speziell für ein bestimmtes Vorhaben hin angepasst sein und es können nicht alle bekannten Algorithmen umgesetzt werden. Dementsprechend liegt ein Computer, welcher eine Praxisbedrohung darstellt noch in ferner Zukunft. Es sollte aber davon ausgegangen werden, dass es ihn eher früher als später geben wird. Grund zu dieser Annahme ist der massive Fortschritt im Bereich Quantencomputing der letzten Jahre. Einer dieser ist der Beweis von QC Überlegenheit gegenüber klassischen Computern, geführt von Google siehe [Aru+19].
Ebenfalls empfiehlt das Bundesamt für Sicherheit in der Informationstechnik die Verwendung von Post-Quanten sicheren Algorithmen für streng geheime Informationen bereits zum jetzigen Zeitpunkt, siehe [Hag20]. Dies wird verständlich im Gesichtspunkt des "Store-Now-Decrypt-Later„ Prinzips, wobei Daten jetzt aufgezeichnet werden, um sie zu einem späteren Zeitpunkt mittels Quanten-Computer zu entschlüsseln. Im folgenden Punkt wird näher auf die tatsächliche Bedrohung durch QC eingegangen. Es ist festzustellen, dass die Bedrohung durchaus real ist, jedoch gerne überspielt wird. In den letzten Jahren entwickelte sich die Quantentechnologie von einem reinen Forschungsgebiet, in ein gesellschaftlich relevantes Thema. Erkennbar in der stetig steigenden Berichterstattung außerhalb der Fachwelt. In naher Zukunft wird Quantentechnologie immer mehr in die IT-Welt eindringen und dementsprechend wird die Frage der Sicherheit gegenüber dieser neuen Technologie immer größer.

1.3 Begründung

Welche Bedrohung durch Quantencomputer bestehen, wird oft vage beschrieben und in Technikzeitschriften ist gerne vom ”Ende der Kryptografie„ die Rede. Diese Aussage ist sicherlich übertrieben, dennoch stimmt es, dass bestimmte Teilgebiete und Algorithmen der Kryptografie durch einen entsprechend leistungsstarken Quantencomputer obsolet werden. Dies ist der Tatsache geschuldet, dass QC bestimmte mathematische Probleme deutlich effizienter lösen können als herkömmliche Computer. Ein Beispiel hierfür ist das faktorisieren großer Primzahlen, wofür es auf klassischen Computern keinen Algorithmus gibt, welcher dies in nutzbarer Zeit bewerkstelligen kann. Jedoch existiert ein Quantenalgorithmus, welcher eben jenes Problem lösen kann. Dieses Szenario existiert für mehrere verbreitete Algorithmen, was die Bedrohung von Quantencomputern durchaus real macht. Was an dieser Stelle aber ebenfalls erwähnt werden muss, ist die Tatsache, dass diese Probleme hauptsächlich asymmetrische Kryptografieverfahren betrifft. Der weitverbreitete symmetrische AES Algorithmus zum Beispiel ist auf Quantencomputer nach jetzigem Stand genauso sicher wie auf klassischen Systemen. Leider sind unter den betroffenen Algorithmen der zwei weitverbreitetsten Kryptosysteme RSA und DSA. Dementsprechend müssen für diese Alternativen gefunden werden, welche auf klassischen und Quantencomputern gleichermaßen sicher sind und den selben Funktionsumfang bieten.
Wann mit der Entwicklung eines QC zu rechnen ist, welcher aktiv Kryptosysteme gefährdet, ist schwer vorauszusagen. Tendenz ist, dass die nächsten 5 bis 10 Jahre noch keine direkte Bedrohung besteht, entsprechend einer Studie des Global Risk Institutes (siehe [MP19]). Folgende Grafik fasst die Ergebnisse der Studie zusammen:

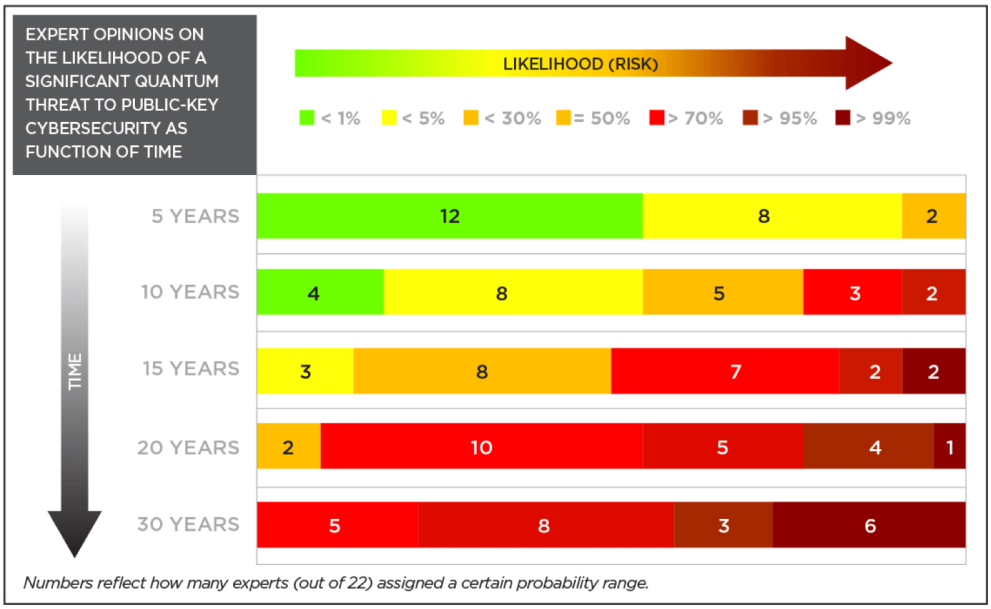


Abbildung 1: Studie zur Bedrohung durch Quantencomputer

Aktuelle Quantencomputer sind, wie bereits angesprochen, noch zu fehleranfällig um beliebige Quanten-Algorithmen auszuführen und müssen speziell auf ihren Verwendungszweck hin angepasst werden. Dennoch ist der Zeitpunkt wo dies nicht mehr der Fall ist, absehbar und daher sollte dieses Thema nicht vernachlässigt werden.

1.4 Zieldefinition

Die These dieser Arbeit ist, dass ein Post-Quanten-Kryptografie existiert welcher in Ada implementiert werden kann und sich mittels SPARK verifizieren lässt. Zum prüfen dieser These werden unterschiedliche PQC Algorithmen untersucht und bei einer erfolgreichen Implementierung verschiedene Tests an diesen durchgeführt. Eine korrekte Umsetzung des gewählten Algorithmuses wird mittels SPARK sichergestellt. Es müssen einige Beweise zur korrekten Implementierung, mittels SPARK geführt werden. Diese sind exemplarischer Natur und müssen nicht auf das ganze Programm ausgeweitet werden. Es sind keine speziellen Zielsysteme zu unterstützen. Die Anwendung wird für aktuelle Windows Systeme entwickelt (zum aktuellen Zeitpunkt Windows 10 2004). Es existieren keine Bestimmungen zur Programmgestaltung abgesehen von der Wahl der Programmiersprache. Es sind keine Vorgaben zur Effizienz der Implementierung getroffen, generell wird jedoch nach Möglichkeit des zeitlichen Rahmens optimiert. Es sollen verschiedene Test an der Implementierung bezüglich praktischer Eigenschaften durchgeführt werden. Es wird keine Sicherheitsanalyse ausgeführt, relevante Quellen werden diesbezüglich zitiert. Bewertung der Sicherheit findet ausschließlich auf Basis gegebener theoretischer Beweise statt.

2 Grundlagen

2.1 Quanten Computing

Quanten Computer sind eine neuartige Entwicklung, welche als Basis für Berechnungen, nicht wie üblich binäre Transistoren nutzen, sondern auf Qubits basieren, welche zwei Zustände gleichzeitig abbilden können. Diese Zustände sind 1 und 0 analog zum binären Transistor, dies wird erreicht durch eine Quantensuperposition des einzelnen Qubit. Eine Entscheidung für 1 oder 0 wird erst bei der Messung getroffen, wenn die Superposition kollabiert und einen Wert anhand verschiedener Einflüsse annimmt. Man spricht vom Kollaps der Wellenfunktion, diese bildet die Wahrscheinlichkeit eines bestimmten Quantenzustandes über die Zeit ab. Bei einer Messung werden die beiden möglichen Zustände des Systems auf einen reduziert.

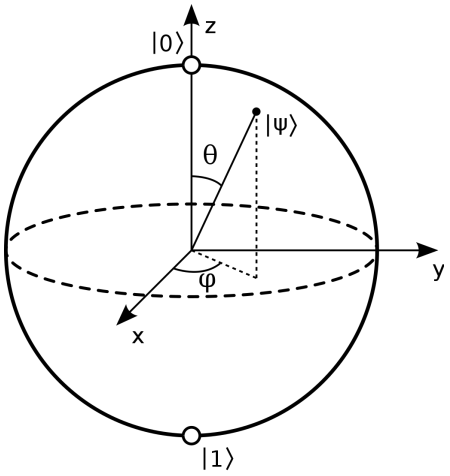


Abbildung 2: Qubit als Bloch-Kugel

Bei dieser Darstellung handelt es sich um eine Bloch-Kugel oder Sphäre, diese wird häufig für Qubits verwendet, ist jedoch nicht die einzige Darstellungsform. Hier erkennt man gut die Eigenschaften eines Qubit beide Zustände 1 und 0 gleichzeitig zu verkörpern. Anstatt von zwei Fixpunkten existieren Wahrscheinlichkeit, welche auf einen Zustand hindeuteten. Diese Wahrscheinlichkeiten können komplexe Zahlen sein. Bis zur Messung wo das System einen festen Wert annimmt, kann sich den Wahrscheinlichkeiten nur bis zu einem gewissen Grad angenähert werden. Nicht jedes Quantenmechanische System eignet sich jedoch als Qubit, es müssen verschiedene Kriterien eingehalten werden, um eine sinnvolle Nutzung zu ermöglichen. Diese beinhalten, dass ein reiner Zustand der Qubits erreichbar sein muss und dass sich einzelne Qubits gezielt messen lassen. David DiVincenzo hat sieben Kriterien für Qubits aufgestellt, siehe [DiV00]. Eines der größten aktuellen Probleme im Bereich Quantencomputing ist das Halten von mehreren Qubits in kontrollierten Zuständen, was eine Skalierung in die Breite stark erschwert. Gleichzeitig müssen QC nicht die gleiche Anzahl an Bits wie ein herkömmlicher Prozessor besitzen, denn mit n Qubits lassen sich 2^n Zustände gleichzeitig abbilden.

Diese Eigenschaft basiert auf der Quantensuperposition in welcher sich die Qubits vor der Messung befinden. Die tatsächliche Umsetzung eines solchen Qubits basiert auf verschiedenen Technologien wie zum Beispiel Ionen in Ionenfallen oder Superconducting quantum interference devices. Unabhängig von der Technologie sind jedoch alle aktuellen QC NISQ Computer. Dies bedeutet sie sind nur für sehr spezielle Probleme geeignet und besitzen eine zu hohe Fehlerrate um sie breitflächig einzusetzen (siehe [Pre18]). Wann QC die nötige Fehlertoleranz besitzen wurde bereits in 1.3 besprochen und ist Abbildung 1 entnehmbar.
Wie stellt denn nun ein Computer auf dieser Basis eine Bedrohung für moderne Kryptografische Verfahren da?
Es existieren Algorithmen, welche die zugrundeliegenden mathematischen Probleme einiger Kryptoverfahren lösen. Diese sind jedoch auf klassischen Computern nicht effizienter als ein Brute-Force Angriff. Auf einem Quanten-Computer lassen sie sich aber deutlich effizienter ausführen, beziehungsweise sie sind nur mit ihm umsetzbar. Der wohl bekannteste dieser Algorithmen ist Shors Algorithmus ([Sho94]), mit welchem sich effektiv große Primzahlen faktorisieren lassen. Dies ist die Basis von RSA, dem wohl verbreitetsten asymmetrischen Kryptosystem. Ein weiterer ist Gideons Algorithmus mit welchem es möglich ist die Zeit zum Lösen verschiedener Kryptosysteme zu halbieren. Dies lässt sich jedoch mit vergrößern der Schlüssel kontern. Shors Algorithmus bedroht RSA und DES, aber nur den Kernalgorithmus, indem er eine effiziente Möglichkeit bietet große Zahlen zu faktorisieren.

2.2 Betroffene Systeme

Betroffene Systeme sind solche, für welche es einen Quanten Algorithmus gibt, welcher sie in effizienter Zeit lösen kann. Dies muss nicht einer Echtzeitentschlüsselung entsprechen, weil Daten retroaktiv nach dem Aufzeichnen entschlüsselt werden können. Wie bereits zuvor etabliert, erlaubt Shors Algorithmus das Lösen sämtlicher Systeme, welche als Basis das Problem der Faktorisierung haben. Auch für Pell's Gleichung ist ein effektiver Quantenalgorithmus bekannt (siehe [Hal07]), somit sind auch solche Systeme auf dessen Gleichung gefährdet. Für einen kurzen Überblick zeigt folgende Tabelle sämtliche Systeme, welche definitiv durch Quantencomputer obsolet werden (entnommen [BBD09] S.16 Tab. 1).

Kryptosystem	Gelöst durch Quantencomputer ?
RSA Asymmetrisches Kryptosystem	Gelöst
Diffie-Hellman Schlüsselaustauch	Gelöst
Elliptische Kurven Kryptografie	Gelöst
Buchmann-Williams Schlüsselaustausch	Gelöst
Algebraischer Homomorphismus	Gelöst
McElice Asymmetrisches Kryptosystem	Noch nicht gelöst
NTRU Asymmetrisches Kryptosystem	Noch nicht gelöst
Gitter-basierte Asymmetrisches Kryptosystem	Noch nicht gelöst

Abbildung 3: Übersicht Betroffener Kryptosysteme

Gelöst entspricht hierbei, dass ein effizienter Quantenalgorithmus bekannt ist, welcher das Basisproblem des Algorithmus lösen kann. Es bedeutet, dass er Grundlegend "gebrochen„ ist, jedoch trifft dies keine Aussage über den konkreten Anwendungsfall. Abhängig von der Implementierung ist es für einen Angreifer leichter oder schwerer seinen Quantencomputer zu nutzen. Bei Hybriden Implementierungen ist eventuell nur ein Teil gebrochen, was bei weitem nicht optimal aber noch nicht "Worst-Case„ ist. Hybride Implementierungen sind hierbei solche, wo zwei oder mehrere Kryptosysteme hintereinander oder im Einklang genutzt werden. Ein aktueller Ansatz sieht vor, Post-Quanten Algorithmen als zusätzliche Lösung in bestehende Kryptosysteme einzubinden um so zukunftssicher zu sein. Optimal wäre ein PQ Kryptosystem mit gleichen oder fast gleichen Eigenschaften zu gängigen Systemen.

2.3 Post-Quanten-Kryptografie

Nachdem die betroffenen Kryptosysteme näher beleuchtet wurden, stellt sich natürlich die Frage, welche Systeme resistent gegen klassische und Quanten Computer sind? Tatsächlich sind dies nicht nur einzelne Systeme, sondern ganze Kategorien (vergl. [BBD09] S. 1-2).

Symmetrische Kryptografie

Diese Systeme sind bereits weit verbreitet und im Einsatz. Der wohl bekannteste Vertreter ist AES. Der Nachteil eines symmetrischen Kryptosystems ist, dass ein gemeinsames Geheimnis notwendig ist für eine sichere Kommunikation. Zudem ist eine Verifizierung des Chiffrats, wie bei asymmetrischer Kryptografie, nicht möglich ist. Da symmetrische Kryptosysteme eine sehr umfangreiche Abdeckung im Bereich der Fachwelt erfahren wird in dieser Arbeit nicht weiter auf sie eingegangen. Es ist anzumerken, dass nicht alle symmetrischen Systeme von sich aus Postquanten sicher sind, sondern das die etablierten Standards als sicher eingeschätzt werden. Es ist zu vermuten, dass Quanten-Algorithmen existieren, welche etablierte Verschlüsselungssysteme vereinfachen können und eine Reduktion der Zeit zum entschlüsseln herbeiführen. Dies hätte größere Schlüssel zur Folge, was jedoch kein Problem darstellen würde, sollte sich das Mooresches Gesetz weiterhin als wahr erweisen.

Hash basierte Kryptografie

Basiert auf Hashfunktionen, welche oft in Baumstrukturen angeordnet werden. Wie beim wohl bekanntesten Vertreter dieser Kategorie dem Merkle Signatur Verfahren. Hauptsächlich werden diese Systeme für die Signatur eingesetzt, jedoch ist eine vollständige Nutzung als asymmetrisches Schlüsselsystem denkbar. Ein Problem ist die Notwendigkeit, für jede Verschlüsselungsoperation einen Hashbaum aufzuspannen, was sehr rechenintensiv ist und für jeden Anwendungsfall abgesehen von Signaturen ineffizient ist.

Code basierte Kryptografie Ein klassischer Vertreter dieser Kategorie ist McEliece. Dieses Verschlüsselungsverfahren wurde in Konkurrenz zu RSA gesehen. Es wird im Punkt 3.2 näher beschrieben. Im Allgemeinen basieren diese Systeme auf Matrizen, für eine effiziente Berechnung werden in der Regel Goppa-Codes genutzt.

Es ist zwar kein Quantenalgorithmus bekannt welcher diese Systeme effizient lösen kann, aber aufgrund der sehr großen Schlüssel im Verhältnis zu anderen Systemen finden Code basierte Kryptosysteme wenig Anwendung.

Gitter basierte Kryptografie

Bezeichnet Kryptosysteme, welche entweder in ihrer Konstruktion oder Sicherheitsbeweis Gitter nutzen. Sie eigenen sich besonders gut für Kryptografische Anwendungen aufgrund sehr guter Sicherheitsbeweise und effektiver Implementierbarkeit. Für diese Arbeit bilden sie die Gruppe der vielversprechendsten Kandidaten. Zuerst jedoch gilt es zu definieren was ein Gitter ist, es handelt sich dabei um eine Gruppe von Punkten im n-dimensionalen Raum. Formell beschrieben besteht ein Gitter aus n linearen unabhängigen Vektoren $b_1 \cdots b_n \in \mathbb{R}^n$, aus welchen es generiert wird.

$$\lambda(b_1 \cdots b_n) = \sum_{k=1}^{n} x_k b_k : x_k \in \mathbb{Z}$$

Die Vektoren $b_1 \cdots b_n \in \mathbb{R}^n$ bezeichnet man als Basis des Gitters λ. In der folgenden Abbildung wird ein 2-dimensionales Gitter mit 2 Basisvektoren gezeigt.

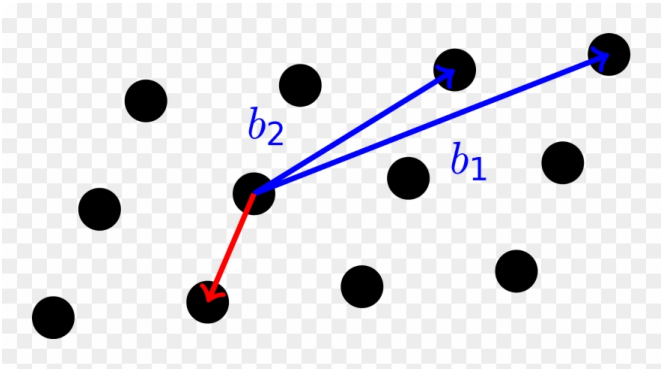


Abbildung 4: Gitter / Kürzestes Weg Problem

Gleichzeitig handelt sich hierbei um eine Darstellung eines der Probleme, welche in Gittern existieren, nämlich das Kürzeste Vektor Problem (SVP Shortest vector problem). Dieses ist eines von mehreren NP-Vollständigen Problemen, welche in Gittern vorkommen. Der Beweis für die Komplexität des Kürzesten Wege Problems findet sich in [Mik98], es ist die Basis für das asymmetrische Verschlüsselungssystem NTRU. Ein weiteres sehr bekanntes Problem welches teilweise auf SVP basiert, ist das "Traveling Salesman Problem„ (Problem des Handlungsreisenden), welches 1930 formuliert wurde. Letzteres eignet sich weniger für Kryptografische Anwendungen zeigt jedoch anschaulich die Komplexität von Optimierungsproblemen in Gittern. Zwei Systeme auf Gitterbasis werden im Abschnitt 3.3 beziehungsweise 3.4 beleuchtet. Allgemein kann angenommen werden, dass es keinen Algorithmus gibt, welcher in Polynomialer Zeit Gitterprobleme approximieren kann (nach [BBD09] S.149).

Multivariante quadratische Gleichungs Kryptografie
Hierbei handelt es sich um ein relativ neues Feld der Kryptografie, bei welchem das Basisproblem darin besteht, nichtlineare Gleichungen über einem begrenztem Feld zu lösen. Es gilt als NP-Vollständig. Üblicherweise bestehen die Schlüssel aus Gruppen von quadratischen Polynomen, welche einen Teil des begrenzten Raums abbilden. Diese Gruppe gilt als besonderes vielversprechend auch starken Quanten-Computern zu widerstehen, was insbesondere auf die rasante Entwicklung der letzten Jahre gestützt wird. Der öffentliche Schlüssel besteht wie bereits erwähnt aus einer Gruppe quadratischer Polynome, definiert als:

$$P = (p_1(w_1, \ldots, w_n), \ldots, p_m(w_1, \ldots, w_n))$$

wobei jedes p_i ein nichtlieares Polynom ist. Sämtliche Koeffizienten und Variablen befinden sich im Feld $\mathbb{K} = \mathbb{F}_q$, bestehenden aus Q Elementen (siehe [BBD09] S. 193 ff). Die Verschlüsselungsfunktion ist üblicherweise die Berechnung dieser Funktionen für einen bestimmten Wert, selbiges gilt für die Verifikationsfunktion. Das Basisproblem bildet sich im Aufwand der Invertierung dieses Feldes ab. Einer der am meist verbreiteten Vertreter dieser Gruppe ist HFEv (hidden field equations) erstmals vorgestellt 1996, siehe [Pat96].

2.4 Ada / Spark

Ada ist eine strukturierte Programmiersprache, welche in den 1970er vom Verteidigungsministerium der Vereinigten Staaten von Amerika in Auftrag gegeben wurde und 1983 erstmalig standardisiert wurde. Sie gilt als eine der ersten standardisierten Programmiersprachen und erhielt mehrmalige Revisionen mit der aktuellsten aus dem Jahr 2012. Diese Version wird auch für diese Arbeit genutzt. Ada findet primär Anwendung in eingebetteten Echtzeitsystemen und Sicherheitskritischen Anwendungen. Dies lässt sich auf die strengen syntaktischen Regeln und die Vielzahl an Validierungswerkzeugen zurückführen. (vergl. [AH20] S. 3 f.). Somit gewährleistet Ada eine höhere Sicherheit gegenüber Laufzeitfehlern und Programmfehlverhalten. Eine Sicherheit gegenüber Fremdeinwirkung und böswilligen Nutzern obliegt aber weiterhin dem Programmierer. Im Englischen wird hier in der Literatur häufig daher zwischen ”safety„ (gegenüber Programmfehlern) und ”security„ (gegenüber Angreifern) unterschieden. Im Deutschen geht dieser Unterschied leicht verloren.
Ada verfolgt verschiedene Paradigmen wie Objektorientierung und einige Elemente von funktionaler Programmierung. Im Kern handelt es sich jedoch um eine Imperative Sprache, ähnlich zu C. Ada ist eine streng typisierte Sprache, daher ist es notwendig neue Datentypen für verschiedene Anwendungsfälle zu definieren. Diese Datentypen können voneinander abgeleitet werden, was erforderlich ist, um verschiedene Datentypen miteinander zu verrechnen. Für diese Arbeit wurde ein klassisch imperativer Aufbau genutzt. Ada selbst wurde gewählt, aufgrund der Möglichkeiten des Frameworks/Sprache SPARK, welche/s im folgenden näher beleuchtet wird. Da die Implementierung nicht ausschließlich in SPARK erfolgen konnte, wurde zusätzlich Ada Code genutzt, welcher explizit nicht von SPARK geprüft wurde.

Eine typische Ada Entwicklung besteht aus mehreren Dateien, da ein modularer Aufbau unterstützt wird und sich dieser zur Strukturierung anbietet. In der Regel sind Codeabschnitte in zwei Dateien aufgeteilt, eine Strukturdatei (*.ads), welche sämtliche Elemente der Bodydatei (*.adb) beschreibt, diese wiederum beinhaltet dann die direkte Implementierung. Es kann sein, dass keine Bodydatei notwendig ist, oftmals dann wenn man nur Konstanten oder eigene Datentypen definiert.

SPARK
Ist zum einen eine Programmiersprache mit Fokus auf funktionale Spezifikation und statische Verifikation, sowie der Name für eine Gruppe an Verifikationswerkzeugen. Sie basiert auf einem Ausschnitt von Ada. Die aktuelle Version, welche für diese Arbeit genutzt wurde ist SPARK 2014 (vergl. [DM20] S.3 f.). Folgende Grafik zeigt das Verhältnis zwischen ADA und SPARK:

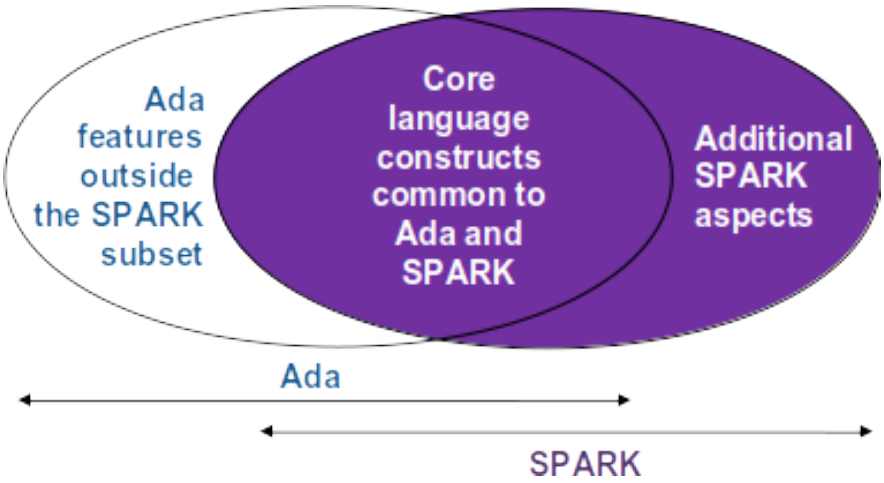


Abbildung 5: Verhältnis von Ada und SPARK

SPARK nutzt verschiedene Ansätze zur Verifikation des Quellcodes, einer davon ist die sogenannte ”Flow Analysis„ (Fluss Analyse). Diese konzentriert sich primär auf Variablen und prüft beispielsweise, ob diese initialisiert sind und tatsächlich Verwendung im Programmablauf finden. Auch gibt es die Option ”Global Contracts„ (Globale Verträge) zu nutzen. Diese definieren, welche Variablen in einem Aufruf modifiziert werden und in welcher Form. Sollte also eine Variable nur als lesende beschrieben sein, kann sie nicht geschrieben werden, ohne einen Fehler zu produzieren. Contracts bieten noch zusätzliche Möglichkeiten als nur die Zugriffsart einer Variable zu bestimmen, sie können auch genutzt werden um Werteräume zu definieren oder Ist und Soll Zustände festzulegen, welche zur Laufzeit geprüft werden können. Definiert werden sie für einzelne Codeabschnitte wie Funktionen oder Prozeduren. Contracts lassen sich als Boolesche Entscheidung verstehen, welche bei einem False eine Exception generiert. Zur Laufzeit eines Codeabschnitts wird die Entscheidung bei jeder Ausführung getroffen, sollte dies aktiviert sein. Contracts sind ein Mittel was SPARK von Ada unterscheidet und können in praktisch jedem Teilbereich der Verifikation eingesetzt werden. Sie bilden die Basis für eines der Grundprinzipien von SPARK nämlich Abstraktion.

Damit ist gemeint, dass die Beschreibung des Codeinhalts (zum Beispiel über einen Contract) unabhängig von der eigentlichen Implementierung ist. Zu diesem Zweck werden können Contracts in der Deklarierungsdatei .ads direkt mit beschrieben werden. Ein weiteres Subset von SPARK ist "Proof of Programm Integrity„ (Beweis der Programm Integrität), welches auf der Fluss Analyse aufbaut, mit dem Ziel, Laufzeitfehler zu verhindern. Zu diesem Zweck werden primär die bereits erwähnten IST und SOLL Zustände für Variablen genutzt, welche in Contracts definiert werden. Abschließend zum "Proof of Programm Functionality„ das Ziel hier ist nicht zu überprüfen, dass Funktionen ausgeführt werden, sondern dass sie auch im Kontext korrekt sind. Es erfolgt also eine inhaltliche Prüfung ob Ergebnisse und das Verhalten von Quellcode der Definition entsprechen. Es ist nicht immer eindeutig all diese Konzepte und Subtypen klar voneinander zu trennen, was auch nicht notwendig ist. Alle bauen aufeinander auf und werden benötigt um ein Programm zu verifizieren. Es ist zum Beispiel von wenig Nutzen beweisen zu können, dass korrekte Berechnungen von einer Funktion ausgeführt werden, wenn dies einen Bufferoverflow erzeugt. Ziel mit SPARK ist es Funktionen soweit zu beschreiben, dass mittels statischer Prüfung eine korrekte Implementation eines Algorithmus festgestellt werden kann. Hierfür ist dann die explizite Umsetzung irrelevant. Damit diese Möglichkeit genutzt werden kann, muss die Beschreibung des Algorithmus allerdings sehr genau sein. Es kann auch nötig sein die Beweisfunktionen mit zusätzlichen Code zu unterstützen, sollten die einzelnen Funktionen zu komplex für einen direkte Beweis sein.

3 Algorithmen Auswahl

3.1 Post-Quanten-Algorithmen

Bei der Suche nach geeigneten Post-Quanten-Algorithmen (ein Algorithmus dessen Sicherheit auch gegenüber QC besteht.) zur Verifikation mit SPARK sollten verschiedene Kriterien beachtet werden. Zum Beispiel der Aufbau, das zugrunde liegende Problem, sowie praktische Eigenschaften wie Schlüsselgröße, Anzahl der Parameter und Implementationsaufwand. Im folgenden werden drei Kandidaten genauer beleuchtet, ein Code-basiertes und 2 Gitter-basierte Verfahren. Die Wahl erfolgte aufgrund der starken Vermutung auf Post-Quanten Sicherheit der einzelnen Systeme, sowie herausstellender Eigenschaften wie langes bestehen oder umfangreiche Dokumentation. Ein System welches nicht näher besprochen wird, dennoch hervorgehoben werden sollte ist der FrodeKEM, basierend auf dem Learning with errors Problem. Dieser befindet sich momentan noch im Standartisirungsprozess der NIST, wird jedoch bereits vom BSI zur Verwendung in den höchsten Geheimhaltungsstufen empfohlen. Er wurde dennoch nicht als Kandidat aufgenommen, da die beiden gewählten Gitter Algorithmen NTRU und LWE vielversprechend sind. Erster ist bereits Standardisiert, zweiter simpler vom Aufbau und basiert auf dem selben Problem wie FrodoKEM. Gegen McEliece konnte er nicht eingetauscht werden, damit nicht ausschließlich Gitter-basierte Algorithmen untersucht werden. Die folgenden Betrachtungen sind unvollständig und dem Vorhaben untergeordnet den Algorithmus zu implementieren. Im falle das sich die Arbeit an einem System als zu zeitaufwendig erweist wird es nicht weiter verfolgt. Die genauen Definitionen lassen sich den jeweils angegebenen Quellen entnehmen.

3.2 McEliece

Der Klassiker unter den Post-Quanten Kandidaten ist McEliece. Entwickelt bereits 1978 von Robert J. McEliece und beschrieben in [McE78], war dieser Algorithmus oft als Kontrahent zu RSA dargestellt. Im Vergleich zu diesem bietet er jedoch einige Nachteile, weswegen er weitaus weniger bis kaum Anwendung über die Jahre erfahren hat. Der größte dieser Nachteile ist wohl die Schlüsselgröße, welche bei 128 Bit Sicherheitsniveau rund 1 MB beträgt. Aufgrund der deutlich besseren Effizienz bleibt RSA daher die zu favorisierende Alternative gegenüber McEliece, auch wenn dieser Post Quanten sicher ist.
Aufgrund seines langen Bestehens, wurde dieser Algorithmus bereits oft untersucht und auf seine Sicherheit geprüft. Daher kann er als Sicher betrachtet werden, auch ist er von NIST standardisiert worden. Als Schlüssel dienen binäre Goppa Codes. Definiert sind diese als:

$$G \in GF(2^m) : \vec{a} \in \mathbb{F}_{2^m}^n \wedge a_i \neq a_j$$

Wobei das zugrunde liegende Problem von McEliece zum "General Decoding Problem„ gehört und daher als NP-Vollständig gilt. Dieser Algorithmus war die erste Anlaufstelle bei der Frage nach einem asymmetrischen Kryptosystem mit Post-Quanten Sicherheit, aber aufgrund seines Alters wurde er schon oftmals betrachtet und die Suche war eher auf neuere weniger verbreitete Systeme ausgerichtet.

Die Vor- und Nachteile von McEliece sind allgemein bekannt weswegen er auch nicht weiter verfolgt wird in dieser Arbeit. Er stellt jedoch eine gute Alternative zu neuen Entwicklungen da, als System, auf welches im Zweifelsfall zurückgegriffen werden kann. Nicht als optimale Lösung, jedoch als praktisch nutzbare und bewiesenermaßen sichere Variante.

3.3 NTRU

NTRU ist ein Gitterbasiertes asymmetrisches Kryptosystem, auf Basis des Kürzesten Wege Problems (SVP- shortest vector problem). Es wird in zwei verschiedenen Varianten beschrieben, abhängig vom Anwendungsfall. Zum einen NTRUEncrypt zur Verschlüsselung von Daten, welches in dieser Arbeit behandelt wird, zum anderen NTRUSign, welches für Signaturen genutzt wird. NTRU wurde erstmals in [HPS98] beschrieben. Im Jahr 2008 wurde der Algorithmus vom Institute of Electrical and Electronics Engineers standardisiert. Das SVP Problem kann wie folgt beschrieben werden.
Ein Vektorraum V und eine Norm N seien gegeben für ein Gitter L, mit dem Ziel den kürzesten nicht null Vektor in V zu finden, gemessen mit N in L. In anderen Worten wird v gesucht, sodass $N(v) = \lambda(L); v \in V, v \neq 0$.
Momentan existiert kein Beweis der Sicherheit für NTRUEncrypt, zwar ist das zugrunde liegende Problem NP-Vollständig, aber daraus lässt sich nicht die Sicherheit des Algorithmus schließen. Die Bewertung der Sicherheit erfolgt anhand dem zur Zeit besten Angriff auf NTRU, zu finden unter [How07]. Als Antwort auf diesen Angriff wurden Parameter Vorschläge erstellt welche bis zu 256 BITs Sicherheit gewährleisten sollen, bei einer Schlüsselgröße von rund 8000 - 120000 BITs. Diese Parameter sind bereits in den IEEE Standard aufgenommen und unter [BBD09] S. 171 näher beschrieben. Eine Implementierung von NTRUEncrypt kann jedoch deutlich Speicherintensiver sein, aufgrund der Funktionsweise der Ver-Entschlüsselungsfunktion, welche große Matrizen multiplizieren. Auch setzt dies einen entsprechend Leistungsstarken Prozessor und optimierte Implementation vor raus um in Echtzeit zu operieren.
Der Algorithmus ist parametrisiert durch die Primzahl n, q nach Empfehlung eine Potenz von zwei und p einer kleinen Ganzzahl, oftmals drei. Der private Schlüssel besteht aus 2 Vektoren (f, g), und aus ihnen wird der öffentliche Schlüssel (h) abgeleitet. NTRUencrpyt ist ein durchaus vielversprechendes System, welches praktisch nutzbar ist und eine starke Vermutung auf Post-Quanten Sicherheit bietet. Das fehlen eines Sicherheitsbeweises ist aber als Nachteil zu betrachten.

3.4 Learning with Errors

Der dritte und vielversprechendste Kandidat ist das LWE (Learning with Errors) asymmetrische Kryptosystem, erstmals beschrieben von Oded Regev in [Reg05] und im Detail beleuchtet in [Reg10]. Letzteres ist ebenfalls Basis für den Implementierungsversuch, sowie die folgende Verifikation mit SPARK. Basis für dieses asymmetrische System ist das LWE Problem parametrisiert durch die Ganzzahlen q, n, sowie die Verteilung χ auf $\mathbb{Z}_q$ und definiert wie folgt: $\mathbb{Z}_q$ beschreibe einen Ring aus ganzen Zahlen modulus q dessen Verteilung χ einer gerundeten Normalverteilung entspricht. $\mathbb{Z}_q^n$ beschreibe eine Gruppe aus n-Vektoren über $\mathbb{Z}_q$, dann existiert eine Funktion $f : \mathbb{Z}_q^n \to \mathbb{Z}_q$. Das LWE Problem beschreibt die Suche nach f mit Eingabe (x,y) für $x \in \mathbb{Z}_q^n$ und $y \in \mathbb{Z}_q$, sodass $y = f(x)$. Durch das addieren eines kleinen Fehlers aus der Verteilung χ wird dieses Problem ausgesprochen hart. Die Verteilung des LWE Problems wird auch ψ_α bezeichnet und lässt sieht wie folgt darstellen (entnommen [Reg05] S.85):

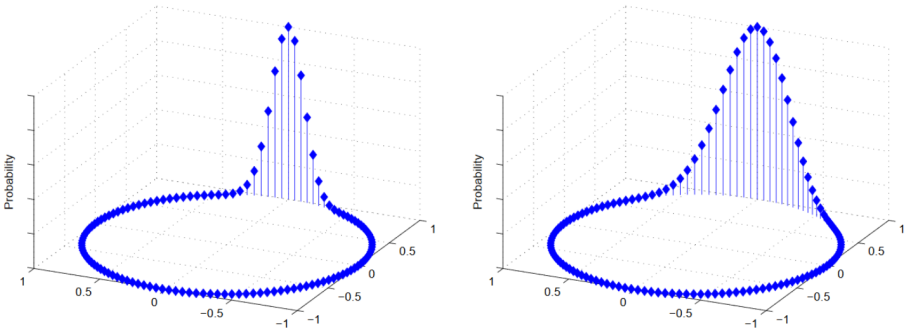


Abbildung 6: Verteilung ψ_α nach [Reg05]

Die schwere des Problems entsteht aus dem Fehler, welcher hinzuaddiert wird, ohne wäre das Problem in rund n Operationen lösbar. Die Verteilung von e soll einer Normalverteilung entsprechen, ohne dass dies weiter spezifiziert wird. Oft wir die Verteilung der verwendeten Zufallszahlen genutzt. Vereinfacht lässt sich das Problem auch als Suche nach einem unbekannten Vektor s in einer Gruppe zufälliger linearer Gleichungen auf s beschreiben. Normalerweise wäre dies als Gleichungssystem zu lösen, aber durch das hinzuaddieren eines kleinen Fehlers (1-3) zu jeder Gleichung wird das Problem schwer. Das Problem ist mindestens NP-Hart mit Vermutung auf NP-Vollständigkeit. Das auf diesem Problem basierende Kryptosystem mit dem gleichen Namen zeichnet sich durch einen theoretischen Sicherheitsbeweis ([Reg05]) und hohe Effizienz aus. Auf einer Basis eines Gitters mit n Dimensionen besitzt der öffentliche Schlüssel eine Größe von $O(n^2)$ und der Algorithmus benötigt $O(n)$ Bit-Operationen für jedes zu verschlüsselnde Bit (nach [BBD09] S.172). Der Beweis der Sicherheit erfolgte gegenüber der Chosen-Plaintext-Attack. Es existiert ein Algorithmus zur Reduzierung des SVP zum LWE-Problem, jedoch handelt es sich hierbei um einen Quantenalgorithmus. Sollte ein effizienter Algorithmus für LWE gefunden werden, würde dies

einen effizienten Quantenalgorithmus für SVP implizieren was momentan sehr unwahrscheinlich scheint (nach [BBD09] S. 173). Algorithmen zum direkten lösen des LWE Problems existieren ebenfalls, der Beste unter diesen operiert jedoch nur in exponentieller Zeit. Vorgestellt wird er unter [BKW03]. Um zur praktischen Umsetzung zu kommen, der LWE Algorithmus besitzt einen zusammengesetzten öffentlichen Schlüssel aus 2 Teilen und einen privaten Schlüssel. Alle drei Komponenten sind Vektoren einer definierten Länge, welche abhängig vom gewünschten Sicherheitsgrad gewählt werden kann. Für die Erstellung der Schlüssel werden Zufallszahlen benötigt, selbstverständlich wären echte Zufallszahlen hier Pseudozufallszahlen vorzuziehen. Leider bestand auf solche kein konsistenter Zugriff für diese Arbeit. Die Werte n, m, l, q, r sind ganzzahlige positive Parameter des Algorithmus. Wobei m die Länge des Schlüsselvektors, q eine Primzahl im Schlüsselraum und r die Anzahl der Elemente, welche aus dem Schlüssel für jede Verschlüsselungsoperation verwendet werden, ist. l und n sind Werte zur Berechnung des Schlüsselraumes und daher die primären Sicherheitsparameter, üblicherweise werden diese gleich gewählt. Der eine Teil des öffentlichen Schlüssels (a) wird gänzlich zufällig gewählt, auch das Geheimnis sollte zufällig sein. Anschließend wird der zweite Teil des öffentlichen Schlüssels (p) wie folgt berechnet:

$$(a, p) = as + e \mod q \in \mathbb{Z}_q^{m*n} * \mathbb{Z}_q^{m*l}$$

Wobei e ein Vektor mit Eigenschaft $e \in \mathbb{Z}_q^{m*l}, \psi_\alpha$ ist, er wird auch als Fehlervektor bezeichnet. Er repräsentiert den bereits erwähnten Fehler, welcher hinzuaddiert wird. Die Verschlüsselungsfunktion setzt eine Gruppe zufällig gewählter Elemente der Schlüssel vor raus, t entspricht r Elementen $\in m$. Verschlüsselt wird abhängig vom Bit für 1

$$(u, v) = \sum_{i \in t} a_i; \frac{q}{2} + \sum_{i \in t} p_i \mod q$$

und für 0

$$(u, v) = \sum_{i \in t} a_i; \sum_{i \in t} p_i \mod q$$

Der Chiffretext besteht erkennbar stets aus 2 ganzen Zahlen, welche als u und v respektive bezeichnet werden. Beide sind für die Entschlüsselung notwendig. Diese funktioniert wie folgt

$$v - (u \times s) \mod q$$

Das Ergebnis dieser ist Null sollte es näher zu 0 als zu $\frac{q}{2}$ sein, ansonsten ist es 1. (siehe [Reg10] S. 14)
Der Algorithmus selber ist vorzugsweise auf Hardware naher Ebene zu implementieren. Bestenfalls mit einem Kompressionsalgorithmus um den Blowup Faktor zu minimieren. Für bessere Verifikation wird er im Zuge dieser Arbeit jedoch auf Nutzerebene eingesetzt.

3.5 Entscheidung

Die Entscheidung, welcher Algorithmus zur Verifikation implementiert werden sollte fiel auf das LWE Kryptosystem nach Regev. Die primären Gründe hierfür werden im folgenden beschrieben. Zusammenfassend waren die einfache Implementierbarkeit, sowie die allgemein guten Eigenschaften gegenüber den anderen Kandidaten ausschlaggebend. Die gute Implementierbarkeit ergibt sich aus der Struktur des Algorithmus, in welcher die komplexeste Datenstruktur ein Vektor mit fester Länge ist. Im Gegenzug benötigt McEliece Goppa-Codes, welche beschrieben und korrekt generiert werden müssen, was eine größere Herausforderung darstellt als die einfachen Vektoroperationen von LWE umzusetzen. Die einfache Struktur erlaubt auch ein leichtes überführen in Programmcode ohne die Notwendigkeit komplexe Zusatzoperationen durchzuführen. NTRU im Gegenzug benötigt für die Generierung seiner Schlüssel komplexe Operationen, welche die Eigenschaften der einzelnen Schlüsselelemente sicherstellen. Zudem bietet LWE einen Sicherheitsbeweis, welcher zwar für McEliece aber nicht für NTRU existiert. Außerdem zeigt sich das Learning with Errors Problem als vielfältig einsetzbar und gut nutzbar, wie an anderen Algorithmen auf dem selben Problem erkennbar ist (z.B. FrodoKEM). Alle Kanidaten gelten als Postquanten sicher, daher kann hier kein Vorteil ausgesprochen werden. Aus ganz praktischer Sicht heraus, ließ sich der LWE am leichtesten Implementieren und damit auch gut untersuchen. Die Entscheidungsgründe nochmals tabellarisch:

Eigenschaft	McEliece	NTRUencrypt	LWE Regev
Erscheinungsjahr	1978	1998	2005
Standardisiert	Ja	Ja	Ausstehend
Theoretischer Sicherheitsbeweis	Ja	Nein	Ja
PQ – Sicher	Ja	Ja	Ja
Schlüsselgröße	ca. 1 MB	ca. 1KB – 15 KB	stark Parameterabhängig 10 KB – 1 MB
Implementierbarkeit (von Grund auf)	Aufwendig – Goppa Codes müssen generiert werden, Matrizenoperationen umsetzen	Mittel – Matrizenoperationen müssen umgesetzt werden	Mittel – Vektoroperationen umsetzen, Generieren der Schlüssel
Bekannte Angriffe	Abhängig vom eingesetzten Code; für Goppa – Codes kein Angriff bekannt	Howgrave-Grahams Hybrid Attacke; Parameter Anpassung nötig	Kein bekannte Angriff

Abbildung 7: Eigenschaftsübersicht

Es muss hervorgehoben werden, dass die Entscheidung mehr aus praktischen Gründen als tatsächlich sicherheitsrelevanten Eigenschaften heraus getroffen wurde. Alle Systeme gelten als sicher und sind Post-Quanten geeignet. Nach der persönlichen Auffassung des Autors, dieser Arbeit ist LWE jedoch am leichtesten umzusetzen. Auch sollte erwähnt werden, dass die Dokumentationen zu LWE gut verständlich und auch für Nicht-Mathematiker nachvollziehbar sind ([Reg05], [Reg10]).

4 Implementierung

4.1 Funktionsdefinition

Die Funktionen, welche das Programm umfassen soll sind primär darauf ausgerichtet die korrekte Implementierung des gewählten Post-Quanten Algorithmus zu prüfen. Sie wurden festgelegt bevor eine Entscheidung für einen Algorithmus getroffen wurde und sind allgemein gefasst. Für jeden Algorithmus wurde ein Implementierungsversuch unternommen, aber nur der LWE Algorithmus nach Regev konnte für sämtliche Funktionen umgesetzt werden. Folgend eine kurze Übersicht der Funktionen:

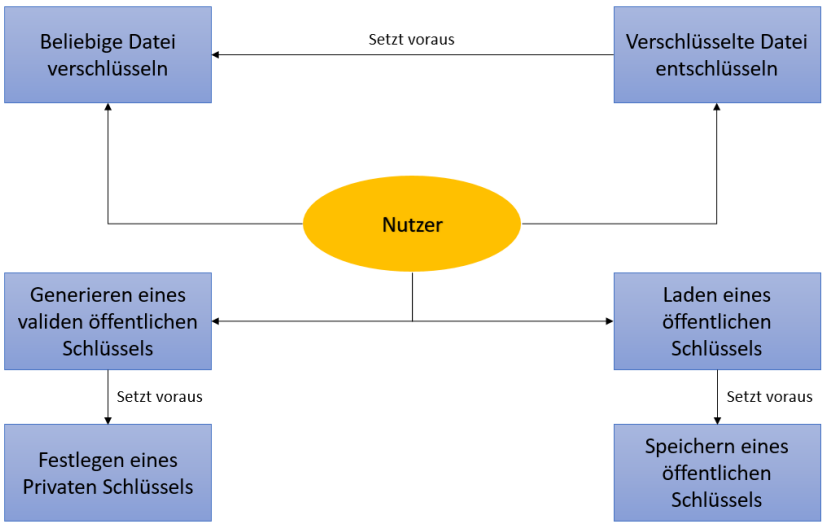


Abbildung 8: Funktionsübersicht

Verschlüsseln einer beliebigen Datei ist selbstverständlich, jedoch sollte der Fokus auf Dateien liegen, da der eingesetzt Algorithmus auf Bit-Ebene verschlüsselt. Daher ist es einfacher die korrekte Funktionsweise zu testen wenn statt einzelner Bits ein Bit-Strom genutzt wird. Das Entschlüsseln schließt sich aus dem Verschlüsseln. Generieren eines validen Schlüsselpaares, setzt sich aus jeweils einem öffentlichen und privaten Schlüssel zusammen. Da das Geheimnis (Privater Schlüssel) benötigt wird um den öffentlichen Schlüssel zu erstellen ist das festlegen eben jenes eine Voraussetzung für die Generierung. Aufgrund der Tatsache, dass die öffentlichen Schlüssel für den eingesetzten Algorithmus große Vektoren sein können, benötigt es eine Funktion diese in Dateien zu schreiben und einzulesen. So kann ein Schlüsselpaar generiert und zu einem späteren Zeitpunkt für die Nutzung in das Programm geladen werden. Einzig das Geheimnis zu einem bestimmten öffentlichen Schlüssel muss der Nutzer sich merken. Zum entschlüsseln einer mit dem öffentlichen Schlüssel verschlüsselten Datei ist nur der private Schlüssel notwendig.

4.2 Ablauf Implementierung

Die Anwendung ist eine reine Konsolenanwendung, da ein grafisches Benutzerinterface als nicht notwendig erachtet wurden. Zu Beginn wurde eine Menüführung erstellt, von welcher aus die verschiedenen Funktionen aufgerufen werde können. Diese Entscheidung entstand aus dem Gedanken, dass dies eine gute Möglichkeit ist sich mit in die Programmiersprache Ada / SPARK vertraut zu machen und dennoch zielführend zu arbeiten. Es bestanden keinerlei Kenntnisse über Ada oder SPARK zu Beginn dieser Arbeit. Mit einer Menüführung entsteht auch eine generelle Vorstellung der einzelnen Use-Cases, diese wurden nicht separat beschrieben, da sie den Funktionen inhaltlich fast gleich wären. Mit einem besseren Verständnis was genau die fertige Anwendung erreichen soll ging es in die Implementierung des Algorithmus. Für alle Kandidaten wurden Implementierungsversuche unternommen, es wurde sich doch schnell für LWE entschieden. Probleme bei der Umsetzung von McEliece war insbesondere die Generierung von Goppa-Codes, bei NTRU wiederum viel es schwer aus der Definition heraus ein funktionalen Programmablauf zu gewinnen. Hierbei zeigte sich besonders, die sich durch alle Algorithmen ziehende Schwierigkeit, aus einer mathematischen Beschreibung einen Programmablauf zu erstellen.
Bei der Implementierung von LWE sind Zufallszahlen unabdinglich. Wenn auch kein Zugriff auf echten Zufall bestand, war der erste Schritt passende Funktionen zu erstellen, welche Pseudozufallszahlen passender Länge und in den passenden Zahlenräumen generieren. Zusätzlich können die dafür verwendeten Zufallsgeneratoren genutzt werden, um den privaten Schlüssel in ein notwendiges Format umzuwandeln. Für das Kryptosystem wäre ein Zufälliger Schlüssel Länge m ideal, jedoch kann sich das kein Nutzer sinnvoll merken. Um dem Algorithmus zu entsprechen und dennoch nutzerfreundlich zu sein, ist der eingegebene private Schlüssel nur ein Seed für die Generierung des tatsächlich verwendeten Schlüssels. Dieser Seed ist eine Ganzzahl im Bereich eines Integers (für Ada 0 - 2^{15}) da sich dies am einfachsten von der Konsole einlesen lässt. Optimal wäre die Möglichkeit ein Passwort aus frei gewählten ASCII Zeichen einzugeben, welche in eine Ganzzahl konvertiert werden, zum Beispiel durch Aufsummierung. Mit der Möglichkeit passende Zufallszahlen zu erstellen, fehlen für den Algorithmus noch passende Parameter, diese wurden [BBD09] Seite 179 Tabelle 3 entnommen.
Mit passenden Parametern und Schlüssel ging es zur Umsetzung der Ver- und Entschlüsselungsoperationen. Da der Algorithmus Bit-weise arbeitet, war der erste Schritt je ein Bit zu verarbeiten. Sicherzustellen, dass dies korrekt abläuft war zu diesem Zeitpunkt sehr schwer, daher wurden die Operationen je 1000 mal ausgeführt und das Ergebnis gespeichert. Bei einem eindeutigen Ergebnis ist davon auszugehen das die Operationen korrekt ausgeführt werden. Auch handelte es sich dabei nicht immer um die gleichen Operationen, da bei jeder Verschlüsselung nur einige zufällig gewählte Elemente des Schlüssel verwendet werden.
Hierbei ergaben sich zwei Möglichkeiten den Ablauf des Algorithmus anzupassen um die Anzahl der notwendigen Schritte zu reduzieren. Es ist möglich die zwei Funktionen, welche für die Verschlüsselung genutzt werden zu einer zusammenzufassen. Die Wahl der Funktion erfolgt abhängig vom Eingangsbit. Wird das Bit hinzugezogen erlaubt es eine Funktion für beide Fälle zu nutzen.

BIT steht im folgenden für das Eingangsbit:

$$v = (\frac{q}{2} \times BIT) + \sum_{i \in t} p_i \mod q$$

Für BIT = 1 entspricht die Funktion der Definition, und bei BIT = 0 entfällt die Addition von $\frac{q}{2}$ was dem Algorithmus entspricht. u ist stets eine Summe aus zufälligen Elementen und daher nicht abhängig vom BIT. Des weiteren kann bei der Entschlüsselung der abschließende Vergleich als $\frac{q}{4}$ zusammengefasst werden. Anstatt als näher zu 0 als zu $\frac{q}{2}$ für das Ergebnisbit Null definiert zu sein. Abgesehen von diesen Optimierungen wurden keine Änderungen am Algorithmus vorgenommen.
Mit LWE selbst implementiert, war der nächste Schritt eine Erweiterung der Eingabe, sodass mehr als ein Bit verarbeitet werden kann. Zu diesem Zweck wurde mit Dateiströmen gearbeitet, es wird je ein Bit gelesen verschlüsselt und anschließend in eine Ausgangsdatei geschrieben. Selbiges vorgehen gilt für die Entschlüsselung einer Datei.

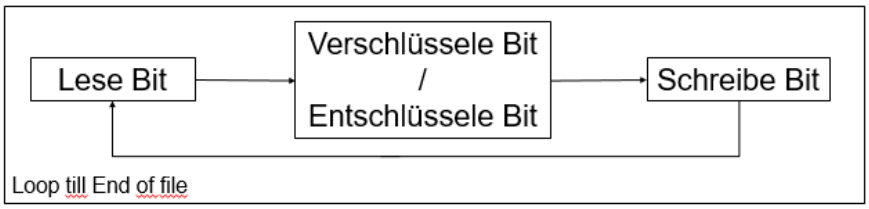


Abbildung 9: Ablauf Ver-Entschlüsselung

Da der Algorithmus Bitweise vorgeht gibt es keine Beschränkungen, was für Dateien verarbeitet werden können. Es wurde die Streaming Bibliothek von Ada genutzt, SPARK bietet generell keine Möglichkeit Datei Ein- und Ausgabe zu verwenden. Die eingesetzte Bibliothek bietet mehrere Vorteile, insbesondere Bitweises arbeiten. Dies ist bei weitem nicht die performanteste Implementierung, da sehr viele Schreib- und Leseoperationen auf der Festplatte ausgeführt werden. Aber es erlaubt das Testen von beliebigen Dateien unbestimmter Größe. An dieser Stelle war es möglich LWE zur Ver- und Entschlüsselung von beliebigen Dateien zu verwenden. Abschließend wurde noch eine Möglichkeit implementiert den öffentlichen Schlüssel in Dateiform zu exportiere und zu importieren. Somit ist es möglich ein Schlüsselpaar mehrfach zu verwenden, und theoretischer Weise einer weiteren Partei den Schlüssel mitzuteilen um vertraulich zu kommunizieren. Aus dem öffentlichen Schlüssel besteht kein Rückschluss auf das Geheimnis, somit kann er auch über ein unsicheres Medium übertragen werden. In diesem Fall sind jedoch weitere Maßnahmen zur Validierung nötig, wie zum Beispiel eine Signatur, um zu bestätigen, dass der Schlüssel auch vom Kommunikationspartner stammt und nicht ausgetauscht wurde. Eine solche Erweiterung ist nicht Teil dieser Arbeit, da es nicht zielführend wäre. Mit der Implementierung abgeschlossen wurde sich auf Tests und Verifikation fokussiert. SPARK war bereits während der Entwicklung mit einbezogen.

4.3 Funktionstests

Um eine korrekte Funktionsweise der Programmfunktionen sicherzustellen, sollen einige Funktionen getestet und die Ergebnisse dokumentiert werden. Folgende Testszenarien wurden festgelegt (angelehnt an Abbildung 8).

Testszenarien	Beschreibung
Öffentlich Schlüssel generieren	Es soll ein valider öffentlicher Schlüssel generiert werden, zu diesem Zweck muss ein Privater Schlüssel gewährt werden. Der öffentliche Schlüssel soll anschließend als Datei gespeichert werden.
Laden eines öffentlichen Schlüssels	Ein valider öffentlicher Schlüssel soll in das Programm geladen werden, Abwesenheit eines solchen produziert eine Fehlermeldung.
Verschlüsseln einer Datei	Eine beliebige Datei wird mittels eines öffentlichen Schlüssels verschlüsselt.
Entschlüsseln korrekt	Eine beliebige Datei wird mittels des korrekten privaten Schlüssels entschlüsselt. Die Datei ist lesbar und unverändert.
Entschlüsseln fehlerhaft	Eine beliebige Datei wird mittels eines inkorrekten privaten Schlüssels entschlüsselt. Die Datei ist unlesbar.

Abbildung 10: Testszenarien

Diese Testszenarien dienen nicht der Verifikation des Algorithmus, sondern sollen zeigen wofür das Programm nutzbar ist und wie die einzelnen Funktionen umgesetzt wurden. Zu diesem Zweck, wird der Ablauf jedes Testszenarios kurz beschrieben und abschließend geurteilt, ob die beschriebenen Funktionen erfüllt werden. Für die einzelnen Tests gibt es jeweils einen Ausgangszustand und Endzustand, welcher beschrieben wird.

Öffentlichen Schlüssel generieren

Ausgangszustand: Programm wird in einem Verzeichnis gestartet, auf welches der Nutzer Schreibzugriff besitzt.

Durchführung: Es wird die Option "Generieren eines Schlüsselpaares„ gewählt. Anschließend wird nach einem privaten Schlüssel in Form einer Zahl von 1 bis 32000 gefragt. Der zweiteilige öffentliche Schlüssel erscheint im Startverzeichnis der Anwendung.

Endzustand: Der öffentliche Schlüssel existiert als Datei im .bin Format. Anlage 1 zeigt den Ablauf.

Laden eines öffentlichen Schlüssels

Ausgangszustand: Programm wird in einem Verzeichnis gestartet, auf welches der Nutzer Schreibzugriff besitzt. Zudem existiert ein öffentlicher Schlüssel entsprechend der Namenskonvention.

Durchführung: Die Option Laden eines öffentlichen Schlüssels wird gewählt. Wenn die zwei Komponenten entsprechend der Namenskonvention vorhanden sind wird der Schlüssel geladen. Andere Szenarien verursachen eine Fehlermeldung.

Endzustand: Der Schlüssel ist nutzbar innerhalb des Programms, siehe Anlage 2.

Verschlüsseln einer Datei

Ausgangszustand: Es befindet sich eine Datei, welche verschlüsselt werden soll im Startverzeichnis des Programms. Es wird ein öffentlicher Schlüssel generiert und damit die Datei verschlüsselt.

Durchführung: Nach der Generierung des Schlüsselpaares wird die Datei übergeben und verschlüsselt. Die Verschlüsselte Datei befindet sich nebst der Eingangsdatei im Startverzeichnis des Programms.
Endzustand: Die Datei wurde verschlüsselt, ein SHA-1 Hash generiert mit dem Microsoft Tool "fciv.exe„ (File Checksum Integrity Verifier) zeigt den Unterschied beider Dateien. Der Versuch die Datei zu lesen ergibt keinen sinnvollen Inhalt. Es ist hervorzuheben, dass die Dateiendung der Ausgangsdatei für den Hashnachweis entfernt wurde. Dies hat den Hintergrund das normalerweise ein ".bin„ angehängt wird um das überschreiben der Eingangsdatei zu vermeiden, was hier jedoch den Hashwert verfälschen würde. Siehe Anlage 3 für einen Bildnachweis.

Entschlüsseln Korrekt

Ausgangszustand: Es existiert eine verschlüsselte Datei und der private Schlüssel ist bekannt. Die Datei wird aus dem vorangegangen Testfall übernommen.
Durchführung: Die Datei wird mittels der Funktion Entschlüsseln und der Angabe des korrekten privaten Schlüssels entschlüsselt. Sie erscheint in korrekter Form (bezüglich Endung) im Startverzeichnis der Anwendung.
Endzustand: Die Datei ist lesbar, der Hashwert ist gleich zur Eingangsdatei. Anlage 4 zeigt das Ergebnis.

Entschlüsseln fehlerhaft

Ausgangszustand: Es existiert eine verschlüsselte Datei und der private Schlüssel ist unbekannt.
Durchführung: Die Datei wird mittels der Funktion Entschlüsseln und der Angabe eines zufälligen privaten Schlüssels entschlüsselt. Sie erscheint in korrekter Form (bezüglich Endung) im Startverzeichnis der Anwendung.
Endzustand: Die Datei ist unleserlich und der Hashwert zeigt eine Veränderung gegenüber der Eingangsdatei (entnommen Testfall 4). Anlage 5 enthält die Darstellung.
Es kann zusammengefasst werden, dass alle Testfälle erfolgreich durchgeführt wurden und sich alle Funktionen wie erwartet verhalten haben. Damit gelten die in 4.1 definierten Funktionen als umgesetzt.

4.4 Verifikation mit Spark

Im folgenden werden nun verschiedene Beispiele gezeigt, wie SPARK in der Anwendung genutzt wird um die korrekte Umsetzung des Algorithmus zu gewährleisten. Hierbei wird nochmals auf die verschiedenen Arten der Verifikation mit SPARK eingegangen, diese sind Flow Analysis, Proof of Integrity und Proof of Functionality. SPARK ist mehr als nur ein Werkzeug um korrekte Programmausführung zu gewährleisten, es ermöglicht statische Beweisführung, welche in der Lage ist die korrekte Umsetzung eines Algorithmus anhand von Beschreibungen zu gewährleisten. So ist es möglich zu bestätigen, dass beispielsweise LWE korrekt implementiert wurde ohne jemals die explizite Implementierung anzusehen. Für diesen Grad der Verifikation, benötigt es jedoch eine sehr umfangreiche formelle Beschreibung des Algorithmus und abhängig von der Komplexität, der einzelnen Operationen Hilfscode, welcher den Beweiser bei seiner Arbeit unterstützt. Leider war es nicht möglich im Zuge dieser Arbeit, diesen Grad zu erreichen, daher wurde exemplarisch vorgegangen und die verschiedenen SPARK Kategorien an einzelnen Beispielen umgesetzt. Die Frage in wie weit SPARK bei einer erstmaligen Implementierung von LWE unterstützen kann, wurde ebenfalls betrachtet. Eine genauere Beschreibung dieses Punktes findet sich in 5.4 Implementierbarkeit. Zu den angesprochenen Kategorien:
Das zeigen Ersterer gestaltet sich schwerer als erwartet, denn anstatt das Definitionen durch den Programmierer als Basis dienen, sind die meisten Prozesse in dieser Kategorie voll automatisch. Zum Beispiel findet eine Prüfung der Variablen auf Initialisierung und tatsächliche Verwendung statt. Es wird geprüft, dass Variablen sinnvoll genutzt werden, also nicht nur geschrieben und anschließend nicht mehr verwendet werden. Ein Beispiel, an welchem sich Flow Analysis zeigt, ist wenn eine Variable definiert und verwendet wird, es jedoch zuvor keine Initialisierung gab. Dies erzeugt eine Exception, außer der erste Schritt ist, der Variable einen statischen Wert zuzuweisen.
Proof of Integrity hingen versucht Laufzeitfehler vorzubeugen. Meist sind dies klassische Bufferoverflows aber es hilft auch dabei, dass Variablen ihre logischen Grenzen nicht verlassen. Eine logische Grenze ist Beispielweise für das aktuell zu verschlüsselnde Bit definiert. Dieses kann in seiner Definition als Bit nur 1 oder 0 sein, jedoch wird im Programmablauf ein Integer zur Speicherung gewählt, da dies einen geringen Konvertierungsaufwand zur Folge hat. An dieser Stelle muss zur Erläuterung kurz erwähnt werden, dass SPARK keinen generellen Datentyp für Bit besitzt und keine implizite Typumwandlung bei Berechnungen erlaubt. Als Beispiel ist es nicht möglich einen Integer mit einem Float zu addieren ohne Konvertierung vorzunehmen, wie es in C möglich wäre. Mittels eines SPARK Contracts lässt sich dennoch sicherstellen, dass nur 1 oder 0 als Bit in die Funktion übergeben werden. Dieser definiert nur als boolsche Funktion, dass der Inhalt der Variable MessageBit nur 1 oder 0 annehmen darf. Eine Darstellung dieses Beispiels befindet sich in Anlage 6, diese beinhaltet auch eine Pseudocode Beschreibung des Contracts. Jegliche Abweichung hiervon würde eine Exception zum Buildzeitpunkt, sowie zur Laufzeit nach sich ziehen. Letzteres ist nur dann der Fall, wenn Laufzeitprüfungen aktiviert sind. Diese Option wird nach Möglichkeit vermieden, da sie zusätzlichen Rechenaufwand verursacht.

Proof of Functionality ist die wohl komplexeste Form der Verifikation, Ziel ist es zu zeigen, dass eine Funktion korrekt ausgeführt wird. Dies gestaltet sich für ein Kryptosystem als anspruchsvoll. Die Ergebnisse bieten keinen Rückschluss auf die ausgeführte Berechnung, daher kann nicht bewiesen werden, dass sie die korrekte Berechnung als Quelle hatten. Wäre es möglich statisch zu beweisen das ein beliebiges Ergebnis aus dem LWE Kryptosystem stammt, ohne Zugriff auf die Parameter und das Geheimnis zu haben, wäre dies eine Verletzung des Zero-Knowledge Prinzips. Die verwendeten Operationen von LWE sind Trap-Door-Funktionen. Somit entfallen einfache Vorgehensweisen und der Beweis korrekter Ausführung muss anhand von allgemeinen Informationen, wie Schlüssellänge oder Schlüsselraum stattfinden. Ein Anwendungsbeispiel hierfür ist die Funktion des Ladens eines öffentlichen Schlüssels, diese ist unabdinglich, da der Schlüssel wiederverwendbar und übertragbar sein soll. Entsprechend der Definition muss der Schlüssel eine bestimmte Länge und jedes Element im Schlüsselraum sein. Zudem muss für die zweite Komponente p jedes Schlüsselelement $p = p \mod q$ erfüllen. Dies kann mittels Postconditions sichergestellt werden. Dieses Szenario ist in Anlage 7 dargestellt. In dieser Darstellung befindet sich ebenfalls Pseudocode, wie dies umgesetzt wurde.

5 Auswertung

5.1 Performance Tests

Zur Auswertung des Algorithmus werden im folgenden verschiedene Tests durchgeführt. Begonnen wird hierbei mit einem Performance Test, welcher das Verhalten des Algorithmus mit steigender Dateigröße über die Zeit beurteilt. Der Test wird zuerst mit den Empfohlen Parametern nach [BBD09] Seite 179 Tabelle 3 durchgeführt und anschließend mit reduzierten Werten.

Parameter	Wert
n	192
l	192
m	1500
q	8191
r	5

Abbildung 11: Empfohlene Parameter

Der Zweck hiervon ist es bei weiterhin bestehender Sicherheit, eine Verringerung des zeitlichen Aufwands zur Verschlüsselung zu erreichen. Zusätzlich muss erwähnt werden, dass die Implementierung rein imperativ abgearbeitet wird und keine Optimierung für Mehrkern Systeme oder multiple Threads vorgenommen wurde. Dies hat den Hintergrund, dass es zeitlich nicht mehr möglich war eine saubere Umsetzung zu erreichen, da wie bereits beschrieben mehrere Implementierungsanläufe getätigt wurden. Dies gilt ebenso für sämtliche folgende Tests. Die für diesen Test genutzten Dateien sind im .bin Format. Formate, welche Komprimierung vornehmen, typischerweise Bildformate wie png oder jpeg führen zu ungleichen Ergebnissen. Die genutzten Dateien wurden mittels der Microsoft Powershell erstellt, der verwendete Befehl findet sich in Anlage 8. Folgend die Ergebnisse des Performance Tests mit empfohlenen Parametern:

Zeit in Sekunden	Dateigröße in Kilobyte
0,36	10
3,70	100
16,11	500
21,83	1.024
42,45	2.048
108,56	5.120
151,84	7.168
215,21	10.240

Abbildung 12: Performance Empfohlene Parameter

In Anlage 9 befindet sich die grafische Darstellung als Liniendiagramm. Sämtliche Ergebnisse sind der Mittelwert aus 3 Messungen mit den selben Parametern. Es lässt sich eine lineare Entwicklung feststellen, jedoch ist die Steigung selbst recht stark, was bedeutet, dass größere Dateien mit dieser Implementierung nicht verschlüsselt werden sollten.

Die Grenze für den Einsatz sollte für einen ungestörten Arbeitsfluss bei einer optimierten Implementierung 10 Megabyte nicht überschreiten. Geringe Dateigrößen lassen sich jedoch sehr schnell verschlüsseln. Um das Verhalten der optimalen Parameter zu prüfen, wurde in einer zweiten Ausführungsrunde, der selbe Test erneut durchgeführt. Diesmal jedoch mit halbierten Parametern, was eine Verringerung bis zu einer Halbierung der Verschlüsselungszeit nach sich ziehen sollte. Anlage 10 zeigt die Ergebnisse, welche durchaus überraschten, da trotz geringer Schlüsselgröße und Länge die allgemeine Performance schlechter ist als mit optimierten Parametern. Daraus lässt sich schlussfolgern, dass die Parameterwahl auf einer mathematischen Basis stattfinden muss. Eine Anpassung der Parameter sollte zu erst untersucht werden, bevor eine praktische Anwendung stattfindet, sodass unerwartetes Verhalten vermieden wird.

5.2 Blow-UP Faktor

Der Blow-UP Faktor bezeichnet das Verhältnis der Größe des Eingabewerts zum Ausgabewert. Er wird hier wie die Performance dargestellt. Von vornherein ist klar, dass große Dateien sehr stark anschwellen mit LWE, aufgrund der Verschlüsselung von je einem Bit zu 2 Ganzzahlen. Der Algorithmus ist als asymmetrischer auch mehr zur Signatur als zur Dateiverschlüsselung vorgesehen, dennoch sollte der Blow-UP Faktor betrachtet werden und insbesondere sein Verhalten bezüglich Parameter Anpassung. Unter den selben Parametern wie in Abbildung 9 ergeben sich folgende Ergebnisse:

Input in KB	Output in KB	Output in MB
10	1.280	1,25
100	12.800	12,5
500	64.000	62,6
1.024	131.072	128
2.048	262.144	256
5.120	655.504	640
7.168	917.504	896
10.240	1.310.720	1.280

Abbildung 13: Blow-UP Faktor empfohlene Parameter

In Anlage 11 befindet sich die Darstellung als Liniendiagramm. Es zeigt sich ein rapider Anstieg der Ausgangsdateigröße, was einem starken Blowup-Faktor vom 128 Fachen der Eingangsgröße entspricht. Die Entwicklung ist nicht linear und nähert sich eher x^3 an. Somit ist der Algorithmus in momentaner Form nicht zur Dateiverschlüsselung geeignet. Hier zeigt sich wohl der größte Optimierungsbedarf, die Größe des Blow-UP Faktors muss für einen praktischen Einsatz drastisch reduziert werden, beispielsweise mittels eines Kompressionsverfahrens. Hierbei lässt sich auch darauf verweisen das in der ersten Beschreibung von LWE die Empfehlung gegeben wird den Algorithmus Hardwarenah zu implementieren und direkt auf einen Kompressionsalgorithmus aufzusetzen. Auf diese Betrachtung der optimalen Werte folgte ein Vergleich mit den halbierten Werten gleich zum Performance Test.

Interessanterweise zeigt sich keinerlei Veränderung beim Blow-UP Faktor wenn die Werte reduziert werden. Dies lässt sich wohl darauf zurückführen das weiter hin 1 Bit durch 2 Ganzzahlen ersetzt wird und die genutzte Streaming Bibliothek beide Werte in gleich große Blöcke schreibt.

5.3 Schlüsselgrößen

Die Schlüsselgröße wird nicht wie die beiden vorherigen Punkte mittels eines Liniendiagramms untersucht. Das hat den Hintergrund das sie statisch ist und nur von den verwendeten Parametern, nicht aber vom Geheimnis abhängt. Das Geheimnis ist falls nicht zufällig generiert selbstverständlich vom Nutzer abhängig. Der wohl einflussreichste Parameter für die Schlüsselgröße ist m, dieser Bestimmt die Länge des Schlüsselvektors und multipliziert mit n die Größe des Schlüsselraums. Anpassung an diesen Werten führen zu einer Veränderung der Schlüsselgröße. Zwar entspricht ein kleineres m einem kürzeren Schlüssel, jedoch bedeutet dies keine schnellere Verschlüsselung. Es wird stets nur ein Teil des Schlüssels für jedes Bit genutzt, parametrisiert durch r. Generell lässt sich die Schlüsselgröße als $m(n + l)log(q)$ definieren, mit Logarithmus zur Basis 2 (siehe [BBD09] S.174). Natürlich handelt es sich hierbei eher um einen Richtwert, da abhängig von Dateisystem und Speicherart die Schlüsselgröße variieren kann. Bei den verwendeten Parametern entstehen Schlüssel mit ca. 12KB Größe pro Schlüsselpaar.

5.4 Implementierbarkeit

Vor einer allgemeinen Beschreibung der Implementierbarkeit, wird kurz betrachtet in wie weit SPARK zu einer korrekten Umsetzung des Algorithmus beigetragen hat. Zuerst sollte die Frage beantwortet werden, "Was können wir mit SPARK garantieren?„. Die Antwort darauf ist natürlich Anwendungsfall abhängig und für das gegebene Szenario eines Verschlüsselungssystems wie folgt. Es kann stets garantiert werden, dass sämtliche Rechenschritte innerhalb ihrer Grenzen sind, es können logische Grenzen definiert werden, es kann eine Prüfung anhand definiertet Regeln stattfinden und abhängig von deren Beschreibung kann eine korrekte Implementierung sichergestellt werden. Abgesehen davon ist SPARKs strenger Compiler gut darin klassische Programmierfehler, wie uninitialisierte Variablen oder ungewollte Typumwandlung zu verhindern. Dies ist jedoch alles recht allgemein gefasst und im gegebenen Anwendungsfall konnte wie bereits beschrieben, leider nicht das volle Maß von SPARKs Möglichkeiten ausgeschöpft werden. Dennoch waren SPARKs Eigenschaften bezüglich Datentyp Verarbeitung sehr von Vorteil, insbesondere die Option Datentypen zu vererben und damit passende Datenstrukturen für ein gegebenes Parameterset zu erstellen. SPARKs Proof of Integrity stellte sich für die Entwicklung selbst, als am nützlichsten heraus. Insbesondere das Definieren von logischen Grenzen kann hilfreich sein um den Variablenfluss zu überwachen. Proof of Functionality ist eine Funktion, welche insbesondere Folgeentwicklungen unterstützt und sicherstellen kann, dass Codeanpassungen nicht die zentrale Funktion beeinträchtigen. Mit dieser Frage beantwortet folgt nun die generelle Betrachtung der Implementierbarkeit

Vorab muss erwähnt werden, dass der folgende Abschnitt der subjektiven Einschätzung des Autors entspricht und nicht mit empirischen Daten belegt werden kann. Es ist dennoch meiner Ansicht nach sinnvoll, zu bewerten wie leicht sich die Algorithmen umsetzen ließen. Ein Indiz für die folgenden Beobachtungen könnte die Verbreitung der einzelnen Kryptosysteme sein, es wäre jedoch falsch diese als mehr als eine Korrelation zu betrachten.
Implementierbarkeit scheint vorerst wenig sinnvoll zu diskutieren, da sich sämtliche Algorithmen auf klassischen Computern umsetzten lassen und es auch wurden (mindestens für Ihre Bewerbung bei der NIST). Ziel ist hier jedoch zu hinterfragen, wie leicht dies im praktischen Rahmen möglich ist. Beispielweise lassen sich manche Algorithmen (z.B. RSA) sehr einfach umsetzen, unter der Verwendung von üblicherweise standardmäßig vorhandenen Strukturen. Für andere müssen komplexe Datenstrukturen aufgebaut und verwaltet werden, was mehr Aufwand und Verständnis vom Programmierer erfordert (z.B.Hashbaum Signaturen). Auch muss an dieser Stelle außerhalb des akademischen Rahmens gedacht werden. Durch den rasanten Wachstum der Branche kann nicht erwartet werden, dass jeglicher Entwickler die notwendigen Kenntnisse besitzt um aus mathematischen Definitionen korrekten Programmcode abzuleiten. Daher ist es durchaus als Vorteil zu betrachten wenn ein Algorithmus mit einfachen Operationen umgesetzt wird, welche keine massive Anpassung in gängigen Programmiersprachen erfordern. Alle 3 Kandidaten setzen keine komplexe Struktur voraus aber für NTRU und McEliece müssen jeweils größere Matrizen verrechnet werden, was einen erhöhten Speicheraufwand und steigende Abstraktion erfordert. Zur Erklärung für einen Entwickler kann es notwendig sein den Verlauf des Algorithmus laufend nachzuvollziehen, entsprechend die Transformation durch die einzelnen Schritte mitzuverfolgen. Zum Beispiel aus Debug Gründen oder um eine korrekte Umsetzung zu gewährleisten. Bei beiden Algorithmen wird es jedoch schwer, Zwischenschritte sinnvoll anzuzeigen und auszuwerten. Auch fällt eine Bewertung ob eine Berechnung korrekt durchgeführt wird, ohne umfassende mathematische Vorkenntnisse schwer. Im Gegenzug ist es möglich LWE praktisch aus der Definition heraus Formel für Formel direkt in den Quellcode zu übernehmen, was einer fehlerhaften Implementierung entgegenwirkt. Gleichzeitig besitzt LWE den Nachteil, dass kaum eine Sprache passende Datentypen bereitstellt, um die einzelnen Variablen für den Algorithmus zu speichern. Abhängig von der Sprache, könnte daher das Verrechnen von Nutzer-definierten Datentypen mehr oder minder komplex sein. Die verwendeten Operationen von LWE sind einfach nachzuvollziehen, wird die mathematische Begründung und Herleitung nicht betrachtet. Somit ist es möglich, unter Annahme der Korrektheit, die Funktionsweise von LWE praktisch zu erklären, ohne den thematischen Umfang von Schulmathematik zu überschreiten. Dies wird als Vorteil betrachtet. Diese Einschätzung geht von einer Implementierung ohne Nutzung bestehender Bibliotheken aus. Die Erfahrung, diese Algorithmen von Grund auf selbst zu entwickeln wird betrachtet.

5.5 Bewertung

Eine abschließende Bewertung von LWE unter Betrachtung der untersuchten Punkte. LWE ist ein Kryptosystem mit sehr starker Vermutung auf Post-Quanten Sicherheit, welches sich relativ einfach umsetzten lässt. Eine Verifikation mittels SPARK fällt jedoch schwer, aufgrund der Eigenschaften von Kryptosystemen. Es ist bei kleinen Dateigrößen durchaus nutzbar, die Verarbeitungszeit wächst jedoch rapide in unpraktische Bereiche. Hier kann wie bereits erwähnt mittels Optimierung und Mehrkernanpassung noch nachgebessert werden. Der Blowup-Faktor ist massiv und wächst fasst gleich mit x^3, was für Dateiverschlüsselung absolut Praxis untauglich ist. Für eine Signatur oder sehr kleine Datenmengen ist dies eventuell akzeptabel. Somit zeigt sich das LWE in der Praxis für eine Teilmenge von Anwendungsfällen für Kryptosysteme eingesetzt werden kann. Dies setzt jedoch den Zugriff auf echten Zufall vor raus. Auf Desktop-Ebene ist aktuelle Hardware zu empfehlen oder eine Implementierung auf Protokollebene im Zusammenhang mit einem Kompressionsalgorithmus.

6 Abschluss

6.1 Zusammenfassung der Ergebnisse

Abschließend sollen nochmals die Ergebnisse dieser Arbeit zusammengefasst werden. Es ist eine Anwendung in SPARK und Ada entstanden, welche den LWE Algorithmus nach Ode Regev implementiert und mittels SPARK, teilweise die korrekte Umsetzung und Abarbeitung sicherstellt. Es wurde gezeigt, welche mittel SPARK bietet um dies zu bewerkstelligen. Der Algorithmus wurde umfassend analysiert und bewertet. Die Wahl des Algorithmus erfolgte nach dem Vergleich mehrerer potenzieller Kandidaten und anhand von nachvollziehbaren Parametern. Zudem wurde eine Einführung in die Thematik Postquantenkrytografie gegeben und auf die Bedrohung durch Quanten-Computer nochmals aufmerksam gemacht. Der im Zuge dieser Arbeit entstandener Quellcode ist frei verfügbar, genaueres ist dem Abschnitt 10 Verwertung zu entnehmen.

6.2 Vergleich zur Zielstellung

Es wurde ein PQC Algorithmus erfolgreich implementiert. Er kann und wurde analysiert bezüglich seiner praktischen Eigenschaften. Die genannten Konzepte wurden untersucht. SPARK wurde zur Verifikation eingesetzt. Es konnte keine vollständige statische Beweisführung mit SPARK geführt werden. Der Anteil an SPARK ist geringer als Anfangs gehofft. Der verwendete Algorithmus wurde bewertet. Die Ziele gelten als erfüllt, jedoch hätte SPARK mehr Verwendung finden sollen. Zuviel Zeit musste mit Recherche und Einarbeitung verbracht werden, um eine effektive Entwicklung über das bestehende Maß in zu ermöglichen. Insbesondere das Einarbeiten in eine neue Programmiersprache und die Komplexität des Feldes PQC waren Herausforderungen.

6.3 Aussicht

An letzter Stelle sollt nochmals bedacht werden, wohin diese Arbeit führen könnte. Die Thematik PQC wird in den nächsten Jahren, nach allen Erwartungen weiter an Wichtigkeit gewinnen und mit den rasanten Fortschritten im Bereich Quanten Mechanik, schneller Realität werden als man erwarten würde. Es ist noch eine Weile bis ein tatsächlich praxisrelevanter Quanten-Computer gebaut wird, dennoch sollte man bereits jetzt die notwendigen Vorbereitungen treffen. Aufbauenden auf dieser Arbeit könnte ein Regelwerk erarbeitet werden, welches es erlaubt mittels statischer Prüfung durch SPARK eine korrekte Implementierung des LWE Algorithmus festzustellen. Diese Beschreibung wäre sehr gut geeignet um künftige Entwicklungen zu verifizieren und eine korrekte Umsetzung von LWE sicherzustellen. Des weiteren kann das Learning with errors Problem auf neue potenzielle Kryptografische Anwendung hin untersucht werden. Neue Entwicklungen wie FrodoKEM zeigen, dass auf Basis von LWE verschiedene Kryptosysteme möglich sind.

Abschließend stellt sich bei einer praktischen Umsetzung stets die Frage nach der Wirtschaftlichkeit. Eine Analyse der potenziellen Kosten und Aufwände für die Entwicklung von LWE oder allgemein Post-Quanten sicheren Systemen wäre durchaus vorstellbar. Hierbei könnte die Frage betrachtet werden ob es aus wirtschaftlicher Sicht gerechtfertigt ist, bereits heute auf Post-Quanten Algorithmen zu setzen.

7 Literaturverzeichnis

Literatur

[AH20] Raphaël Amiard und Gustavo A. Hoffmann. “Introduction to Ada”. In: *LEARN ADACore* (2020). URL: https://learn.adacore.com/courses/intro-to-ada/index.html.

[Aru+19] F. Arute u. a. “Quantum supremacy using a programmable superconducting processor”. In: *Nature* 574 (2019), S. 505–510. URL: https://doi.org/10.1038/s41586-019-1666-5.

[BBD09] Daniel J. Bernstein, Johannes Buchmann und Erik Dahmen. *Post-Quantum Cryptography*. Berlin Heidelberg: Springer Science & Business Media, 2009. ISBN: 978-3-540-88702-7.

[BKW03] A. Blum, A. Kalai und H. Wasserman. “Noise-tolerant learning, the parity problem,and the statistical query model”. In: *Journal of the ACM* 50 (2003), S. 506–519. URL: https://doi.org/10.1145/792538.792543.

[DiV00] David P. DiVincenzo. “The Physical Implementation of Quantum Computation”. In: *Progress of Physics* 48 (9-11 2000), S. 771–783. URL: https://arxiv.org/abs/quant-ph/0002077.

[DM20] Claire Dross und Yannick Moy. “Introduction to SPARK”. In: *LEARN ADACore* (2020). URL: https://learn.adacore.com/courses/intro-to-spark/index.html.

[Hag20] Dr. Heike Hagemeier. *CYBER-SICHERHEIT, in BSI-Magazin 2020/01 Mit Sicherheit*. 53175 Bonn: Bundesamt für Sicherheit in der Informationstechnik (BSI), 2020. ISBN: BSI-Mag 20/711-1.

[Hal07] Sean Hallgren. “Polynomial-time quantum algorithms for Pell’s equation and the principal ideal problem”. In: *Journal of the ACM* 54 (2007), S. 1–19.

[How07] N. Howgrave-Graham. “A Hybrid Lattice-Reduction and Meet-in-the-Middle Attack Against NTRU”. In: *Advances in Cryptology - CRYPTO 2007* 4622 (2007), S. 150–169. URL: https://doi.org/10.1007/978-3-540-74143-5_9.

[HPS98] J. Hoffstein, J. Pipher und J.H. Silverman. “NTRU: A ring-based public key cryptosystem.” In: *Algorithmic Number Theory* 1423 (1998), S. 267–288. URL: https://doi.org/10.1007/BFb0054868.

[McE78] Robert J. McEliece. “A Public-Key Cryptosystem Based on Algebraic Coding Theory”. In: *Deep Space Network Progress Report* 42-44 (1978), S. 114–116. URL: https://ntrs.nasa.gov/citations/19780016269.

[Mik98] Ajtai Miklós. “The shortest vector problem in L2 is NP-hard for randomized reductions”. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing* (1998), S. 10–19. URL: https://dl.acm.org/doi/10.1145/276698.276705.

[MP19] Michele Mosca und Marco Piani. “Quantum Threat Timeline”. In: *Global Risk Institute in Financial Services(GRI* (2019). URL: https://globalriskinstitute.org/download/quantum-threat-timeline-full-report-2/.

[Pat96] Jacques Patarin. “Hidden Fields Equations (HFE) and Isomorphisms of Polynomials (IP): Two New Families of Asymmetric Algorithms”. In: *Advances in Cryptology — EUROCRYPT ’96* (1996). Hrsg. von Ueli Maurer, S. 33–48. URL: https://doi.org/10.1007/3-540-68339-9_4.

[Pre18] John Preskill. “Quantum Computing in the NISQ era and beyond”. In: *Quantum* 2 (Aug. 2018), S. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79. URL: https://doi.org/10.22331/q-2018-08-06-79.

[Reg05] Oded Regev. “On lattices, learning with errors, random linear codes, and cryptography”. In: *In Proceedings of the thirty-seventh annual ACM symposium on Theory of computing* (2005), S. 84–93. URL: http://portal.acm.org/citation.cfm?id=1060590.1060603.

[Reg10] Oded Regev. “The Learning with Errors Problem”. In: (2010). URL: https://cims.nyu.edu/~regev/papers/lwesurvey.pdf.

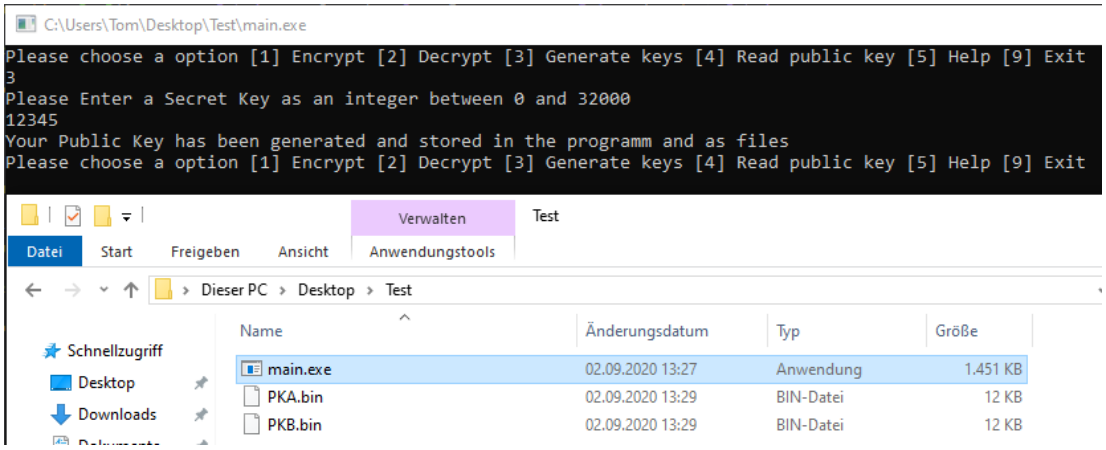
[Sho94] P. W. Shor. “Algorithms for quantum computation: discrete logarithms and factoring”. In: *Proceedings of the 35th Annual Symposium on the Foundations of Computer Science* 35 (1994), S. 124–134. URL: https://doi.org/10.1109/SFCS.1994.365700.

8 Online Quellen

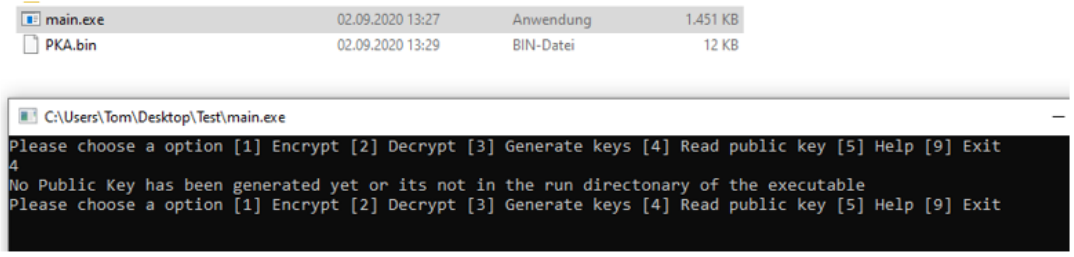
Abbildung 2: Bloch-Sphäre
https://en.wikipedia.org/wiki/Qubit#/media/File:Bloch_sphere.svg (30.09.2020)
Abbildung 4: Gitter Beispiel
https://www.pinclipart.com/picdir/middle/489-4892828_example-shortest-vector-problem-lattice-problem-from-closest.png (31.08.2020)
File Checksum Integrity Verifier https://www.microsoft.com/en-us/download/details.aspx?id=11533 (07.09.2020)

9 Anlagen

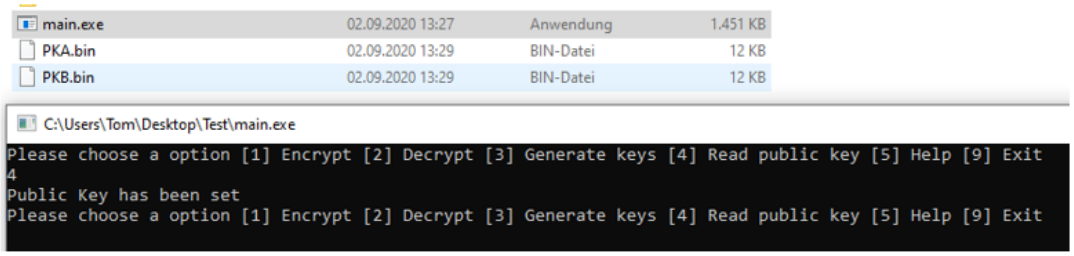
Anlage 1 Testszenario 1



Anlage 2 Testszenario 2



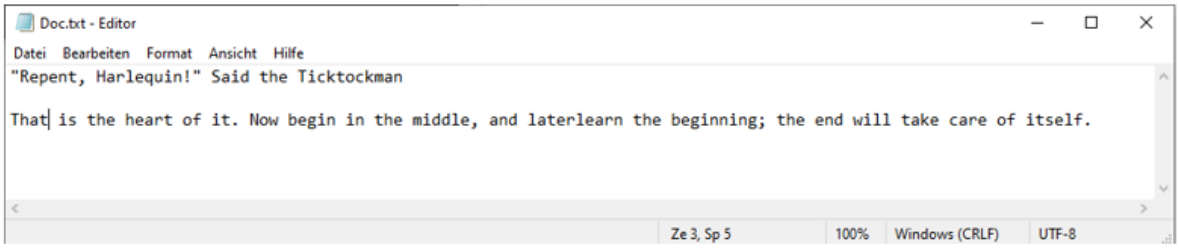
Fehlender Schlüsselteil



Korrekter Ladevorgang

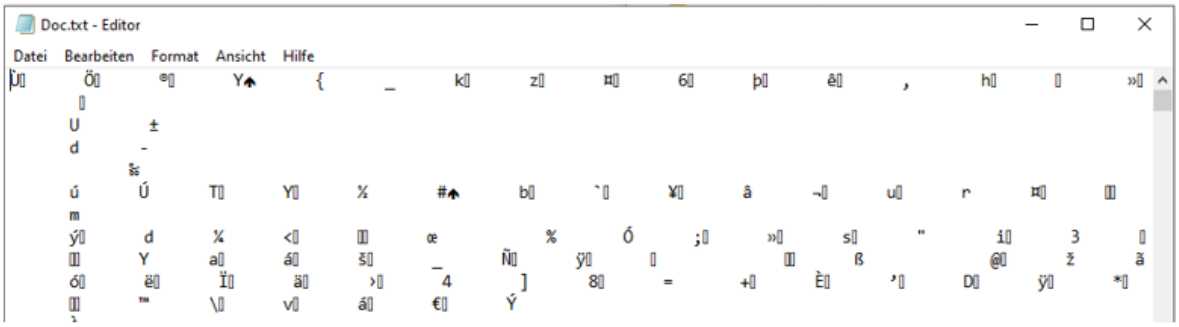
Anlage 3 Testszenario 3

Die Eingangsdatei enthält die Erste Zeile von Harlan Ellison Kurzgeschichte "Repent, Harlequin!" Said the Ticktockman. Der Hash wurde mittels fciv.exe erstellt. Dabei handelt es sich um ein Kommandozeilenprogramm bereitgestellt durch Microsoft.




```
C:\Users\Tom\Desktop\Test>fciv.exe -SAH1 Doc.txt
//
// File Checksum Integrity Verifier version 2.05.
//
3639d4a50654b755fc7a79790569c689 doc.txt
```

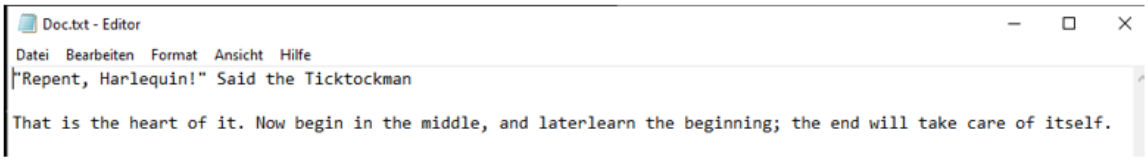
Die Ausgangsdatei ist unleserlich und der Hashwert ist unterschiedlich zur Eingangsdatei.

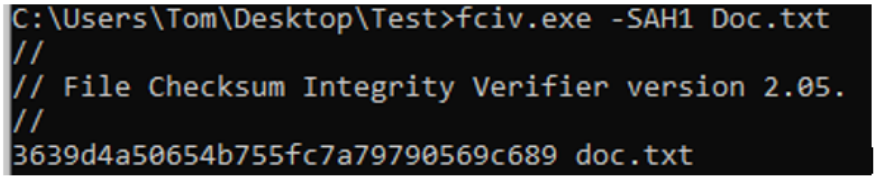



```
C:\Users\Tom\Desktop\Test>fciv.exe -SAH1 Doc.txt
//
// File Checksum Integrity Verifier version 2.05.
//
438f09987fdc2e3fc836bcd6db55a526 doc.txt
```

Anlage 4 Testszenario 4

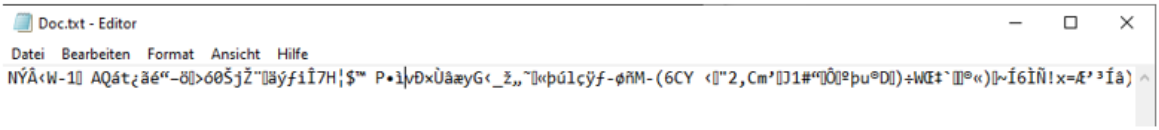
Die Datei ist wieder leserlich und der Hash zeigt eine komplette Wiederherstellung der Eingangsdatei.

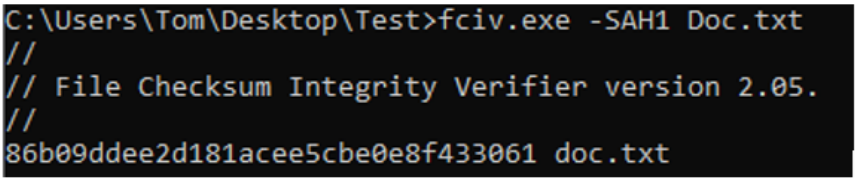




Anlage 5 Testszenario 5

Die Datei ist weiterhin unleserlich und der Hash zeigt eine Veränderung zur originalen Eingangsdatei.





Anlage 6 Proof of Integrity Beispiel

Berechnen des Werts v

$$v = \sum_{i \in m}^{r} \boldsymbol{p}i + \left(\frac{q}{2} * BIT\right) \; mod \; q$$

Wähle Bit zur Übertragung

BIT darf nur 1 oder 0 sein
Typ aber Integer

$$v = \sum_{i \in m}^{r} \boldsymbol{p}i + \left(\frac{q}{2} * BIT\right) = \sum_{i \in m}^{r} \boldsymbol{p}i \; mod \; q$$

Für BIT = 0

Lösung Pre Condition:
BIT => 0 or 1

Pre => (Messagebit = 1 or Messagebit = 0);

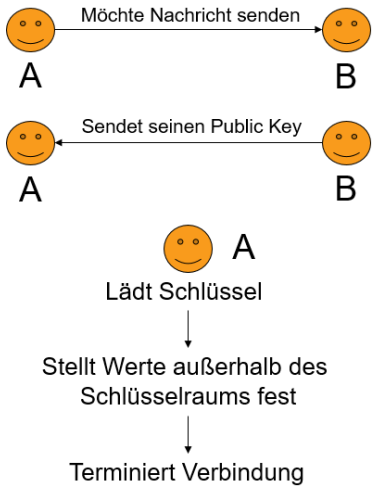
Anlage 7 Proof of Functionality Beispiel

Laden eines Öffentlichen Schlüssels

Voraussetzung: Schlüssel muss feste Länge besitzen UND einzelne Schlüsselwerte müssen im definierten Raum sein

Lösung: Post Condition, welche zum einen Länge zum anderen jedes Element prüft

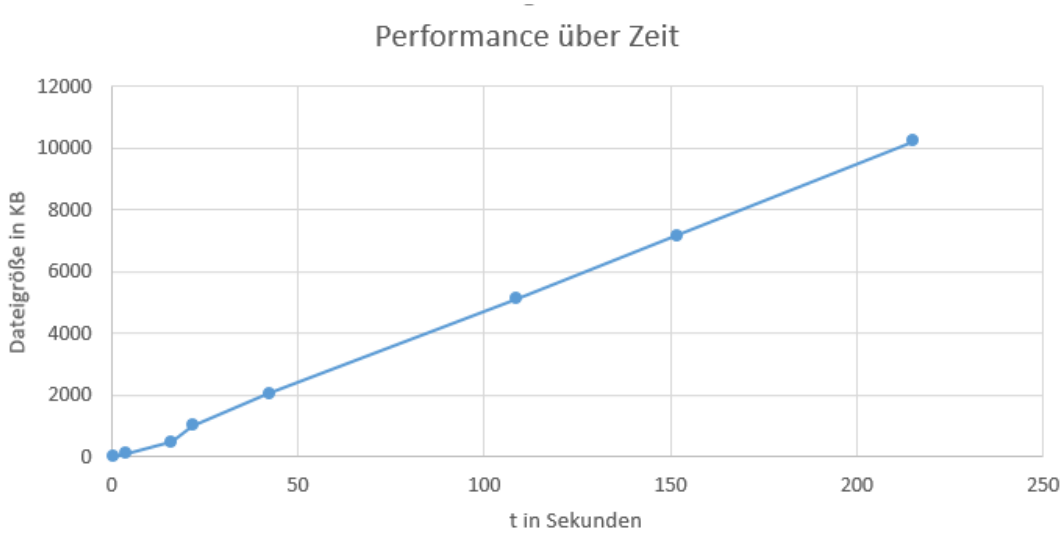
Post => if key is **p** then (for all i in array'range => function'result(i) /= 0 AND function'result(i) = function'result(i) mod q) AND function'result'length = array'length;



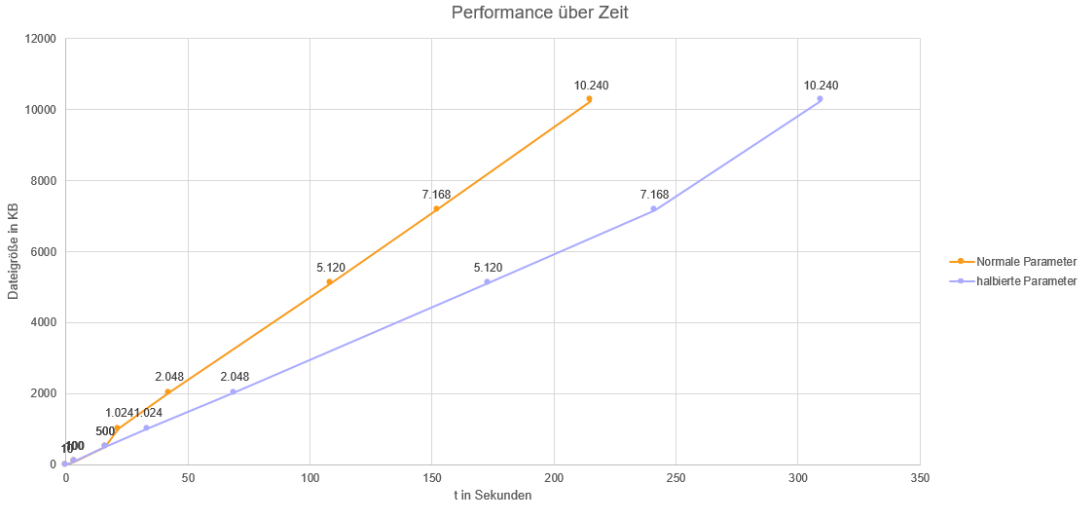
Anlage 8 Dummydateien Powershell

```
$out = new-object byte[] 1048576; (new-object Random).NextBytes($out);
[IO.File]::WriteAllBytes('C:\File.bin', $out)
```

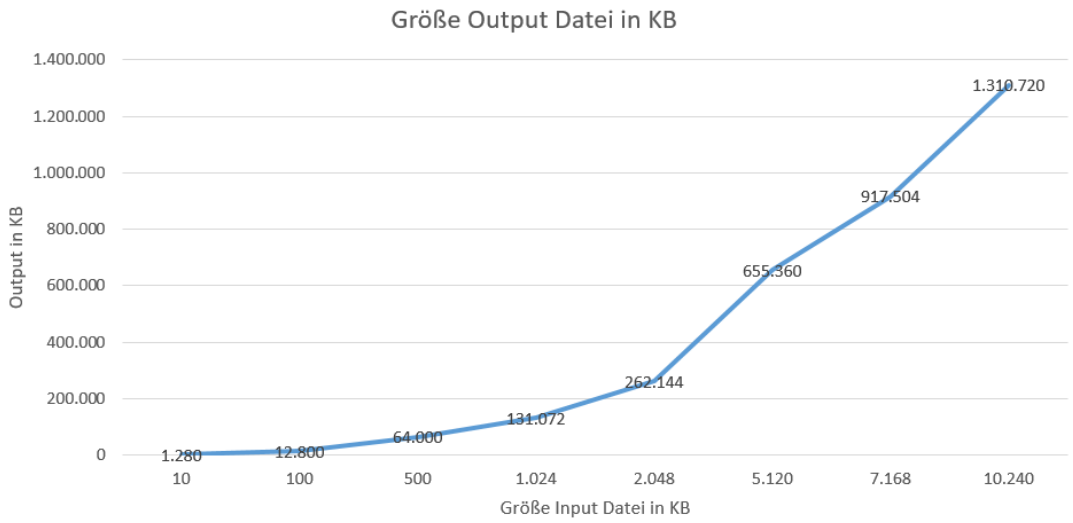
Anlage 9 Performance Liniendiagramm mit empfohlenen Parametern



Anlage 10 Performance Liniendiagramm mit halbierten Parametern



Anlage 11 Blow-UP Faktor Liniendiagramm mit empfohlenen Parametern



10 Verwertung der Anwendung

Sämtlicher im Rahmen dieser Bachelorarbeit entstandener Quellcode ist frei nutzbar und kann nach belieben modifiziert werden. Er wird in GitHub unter folgendem Link bereitgestellt.
https://github.com/TomMarinovic/BachelorLWERegev
Ich verweise an dieser Stelle nochmals auf die Tatsache das sämtliche Entwicklung und Funktionstests unter Microsofts Windows 10 durchgeführt wurden.

11 Selbstständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich reiche sie erstmals als Prüfungsleistung ein. Mir ist bekannt, dass ein Betrugsversuch mit der Note „nicht ausreichend“ (5.0) geahndet wird und im Wiederholungsfall zum Ausschluss von der Erbringung weiterer Prüfungsleistungen führen kann.

Name:	Marinovic
Vorname:	Tom
Matrikelnummer:	43899

Dresden, den 28.09.2020

Unterschrift