LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Gruppe Team 103 – Abgabe zu Aufgabe 504: RSA
Wintersemester 2022/23

Guo Linfeng Özakay Baris Julian Julian

1 Einleitung

Das Praktikum Aspekte der systemnahen Programmierung bei der Spieleentwicklung beschäftigt sich mit dem Programmieren von Prozessen auf niedriger Ebene. Dabei ist es wichtig die Schnittstellen zwischen Hardware and Software effektiv zu nutzen. Außerdem haben wir uns den Umgang mit der Assembly Programmiersprache AArch64 angeeignet und damit verbunden diverse Optimierungsmöglichkeiten für diese. Assembler ermöglicht die Assemblysprache in Maschinensprache zu übersetzten.Der Vorteil an einer derart Hardware-Nahen Programmierung ist, dass man eine viel bessere Leistung für seine Programme erzielen kann, da man angepasste Optimierungen treffen kann, zu welchen ein normaler Compiler aus einer Hochsprache nicht fähig ist. Diese Verbesserung spielt in vielen Bereichen der Informatik eine wichtige Rolle.

Das Praktikum bietet eine Vielzahl an Themengebiete. Eines davon, welchem wir uns gewidmet haben, ist die sogenannte Kryptographie. Kryptographie ist ein essentieller Bestandteil unserer heutigen Kommunikation. Einer der Algorithmen, der unter dem Fachgebiet der Kryptographie zum Einsatz kommt ist der RSA-Algorithmus. Der RSA-Algorithmus wurde im Jahr 1977 zum ersten Mal veröffentlicht und ist nach seinen Erfindern Rivest, Shamir und Adleman benannt. Die Relevanz des Algorithmus hat in der heutigen Zeit nicht nachtgelassen. Er wird in verschiedenen Bereichen wie zum Beispiel bei Banken, Webservern oder E-Mails, für Sicherheit und Datenschutz angewendet. Das Verfahren wird sehr oft für die Verschlüsselung der Kommunikation mit mehreren Teilnehmern genutzt. Dabei kommen zwei Schlüssel zum Einsatz, ein öffentlicher und ein privater Schlüssel. [**?**]

Ein Beispiel dazu: Person A und Person B wollen mit Hilfe des RSA Verfahrens mit einander kommunizieren. Angenommen alle öffentlichen Schlüssel stehen im Internet. A will eine Nachricht an B verschicken und sucht im Internet den öffentlichen Schlüssel von B heraus. Die Nachricht von A wird anschließend verschlüsselt und an B verschickt. Die verschlüsselte Nachricht kann nur von B entschlüsselt werden, da nur er den geheimen Schlüssel kennt.

Ein Schlüssel ist ein Tupel. Der öffentliche Schlüssel ist durch das Tupel (e, N) und der geheimen Schlüssel durch das Tupel (d, N) gegeben. Dabei ist e der Verschlüsselungsexponent und d der Entschlüsselungsexponent. Für die Wahl dieser Variablen gibt es mehrere mathematischen Voraussetzung. Dabei müssen p und q jeweils Primzahlen

sein.

$$N = p \cdot q \tag{1}$$

$$1 < e < \varphi(N) \tag{2}$$

Außerdem muss e Teilerfremd zu φ(N) sein. φ(N) sei die Eulersche φ-Funktion.

$$\varphi(N) = (p-1) \cdot (q-1) \tag{3}$$

Die Variable d lässt sich aus der untenstehende Formel berechnen.

$$d \cdot e \equiv 1 \mod \varphi(N) \tag{4}$$

Nachdem man die Schlüssel generiert hat, kann man die Nachricht m mit Hilfe von e und N verschlüsseln und c ist die verschlüsselte Nachricht. Für die Entschlüsselung wird d verwendet.

$$c{=}m^e \mod N \tag{5}$$

$$m{=}c^d \mod N \tag{6}$$

An Hand eines Beispiels sehen wir wie die Nachricht m = 10 verschlüsselt wird. Wir wählen für N = 1363, e = 3 und d = 859. φ(1363) = 1288 und e ist zudem Teilerfremd. Mit (4) ergibt sich die verschlüsselte Nachricht 1000 = $10^3 \mod 1288$. Mit unserem geheimen Schlüssel wollen wir anschließend entschlüsseln. $1000^{859} \mod 1288$ = m. Mit manueller Kalkulation ist das Ergebnis wie erwartet 10.

Der RSA-Algorithmus besteht aus vier Schritten: Schlüsselgenerierung, Schlüsselverteilung, Verschlüsseln und Entschlüsseln. Unsere Aufgabe ist die Generierung der drei Variablen d, e, N zu implementieren. Wir werden mit der Generation von N anfangen, gehen anschließend darauf ein wie die Variable e gewählt wird und zum Schluss wie man mit Hilfe vom erweiterten Euklidischen Algorithmus aus e und φ (N) unser d erzeugt.

2 Lösungsansatz

2.1 Primzahl Generierung

Für das Generieren von Modulo N braucht man zwei Primzahlen p und q. Aus deren Produkt entsteht wie in der Formel (1) zu sehen N. Es gibt verschiedene Methoden zur Generierung von Primzahlen. Da unsere Primzahlen maxmial 64bit sein können, haben wir die naive Methode gewählt, also wir rechnen eine zufällige Zahl und checken, ob es eine Primzahl ist. Diesen Teil kann man in drei verschiedenen Aspekten untersuchen:

- Generierung der Zufallszahlen:

  Es werden normalerweise kryptographische Zufallszahlgeneratoren empfohlen, es ist aber schwierig in Assembly eines zu implementieren. Deshalb haben wir uns einen kleinen linearen Kongruenzgenerator programmiert. Er nimmt einen Seedwert als Adresse der Eingabe und nutzt die Multiplikations- und Additionswerte aus Donald Knuths Buch "Numerical Recipes". Der Generatorseed ist [0x12345678], der Multiplikator 1664525 und der Umwachs 1013904223. In C-Implementierungen gibt es nur den Unterschied, dass der Seed 0 ist.

  Man muss es aber erwähnen, dass die Pseudo-Zufälligkeit die ganze Algorithmus unsicher macht. In C könnte man einen kryptographischen Generator wie arc4random nutzen, für Vergleichungszwecke haben wir dies aber nicht getan.

  Des weiteren haben wir die Zufallszahl mit 32bit begrenzt, da bei 64bit die erste Zahl mit 50% Wahrscheinlichkeit zwischen 63bit und 64bit ist. Das bringt die Voraussetzung, dass die zweite Zahl relativ sehr klein ausgewählt werden muss. Im Praxis wird das aber wegen kleiner Wahrscheinlichkeit fast nie passieren und damit ist eine sehr lange Laufzeit zu rechnen.

- Die Primecheck-Algorithmus:

  Bei der Primecheck-Algorithmus haben wir auch die kleine Bit-Länge der p und q ausgenutzt und die naive Algorithmus implementiert. Nachdem man für 2 und 3 checkt und die Vielfachen von 2 und 3 eliminiert, geht man alle 6n +-1 bis zum Wurzel der jeweilgen Zahl durch und überprüft ob die Zahl durch den dividierbar ist. Da beim Praktikum in der Haupt-Assembly-Implementierung keine komplexe arithmetische Funktionen wie sqrt(n) erlaubt sind, haben wir einen Umweg gefunden. Statt der Wurzel checkt man, ob die $(6n +- 1)^2$ kleiner als der zu überprüfenden Zahl ist. Man rechnet also $\sqrt{p}$- oder $\sqrt{1}$-mal die $(6n +- 1)^2$, während man die Wurzeln nur ein Mal rechnen könnte. Obwohl klein, sieht man den Unterschied im Teil Performanzanalyse.

- Verhalten bei zusammengesetzten Zahlen:

  Um die Laufzeit möglichst kurz zu halten, akzeptiert das Programm alle Zufallszahlen außer 1. Wenn die Zahl keine Primzahl ist, wird sie faktorisiert und die größte unidentische Zahl wird als Primzahl gewählt. Obwohl das den Zufallsfaktor nicht verletzt, modifiziert es die Wahrscheinlichkeitszuordnung verschiedener Primzahlen. Für uns ist dieses Trade-Off akzeptabel.

2.2 Wahl für e

Der allererste RSA- Algorithmus stammt aus dem Jahr 1977. Diese Variante wurde ursprünglich so implementiert, dass man ein zufälliges d wählt und daraus e generiert. Allerdings ist diese Methode nicht sicher und e würde zu groß sein. Je kleiner bei diesem Algorithmus der Verschlüsselungsexponent e ist, desto schneller ist die Verschlüsselung. Eine große Zahl ist für die Verschlüsselung dennoch performant genug und erfüllt dabei

auch die Sicherheitsanforderungen. Für die Wahl von e muss auf ein paar Voraussetzungen geachtet werden. Der Verschlüsselungsexponent muss größer als eins, und kleiner als φ(N) sein(1). Außerdem muss e teilerfremd zu φ(N) sein.

Damit der RSA- Algorithmus bessere Performance leistet und effizienter ist, muss e eine kurze Bit- Länge und ein kleines Hamming Gewicht haben. Das Hamming Gewicht bei einem String von Bits ist die Anzahl von Einsen. Die Bit Länge gibt die Anzahl an Bits, wenn eine Zahl in Binärschreibweise umgewandelt wird. Um diese Länge einer ganzen Zahl zu bestimmen, wird die folgende Formel verwendet:

$$bitLength(n) = \lfloor \log_2(n) + 1 \rfloor = \lceil \log_2(n+1) \rceil \tag{7}$$

Der Grund für eine Verbesserung der Implementierung bei einem Verschlüsselungsexponent e mit einer kurzen Bit- Länge und einem kleinen Hamming Gewicht liegt an folgenden Eigenschaften:

1. Weniger Multiplikation: Um eine Nachricht zu entschlüsseln, muss man die Nachricht e- Fach mit sich selbst multiplizieren. Daher ist es weniger Rechenaufwand, falls e klein ist und verkürzt somit auch die Berechnungszeit.(3)

2. Weniger Modulo Reduktion: Nach der Berechnung von m^e wird die Operation Modulo N ausgeführt. Wenn e eine kurze Bit Länge hat, so muss weniger Modulo Reduktionen durchgeführt werden. Eine Potenz Modulo Berechnung hat eine Zeitkomplexität von O(log y). Dabei sei y der Exponent.

3. Kleineres Hamming Gewicht: Ein kleineres Hamming Gewicht bedeutet, dass die Zahl in der Bit- Schreibweise mehr Null- Bits hat. Dies spart ebenfalls Rechenaufwand bei der Potenzrechnung und der Modulo Operation.

Die meisten gewählten Zahlen für e sind 65537 und 3. 65537 ist nicht nur groß genug sondern auch Primzahl und kann für die Sicherheit der Verschlüsselung sorgen. Zudem hat diese Zahl ein kleines Hammning Gewicht, in Binärschreibweis 0b10000000000000001. Die kleinste Zahl für e ist die drei. Drei hat ein Hamming Gewicht von zwei, 0b11, und ist somit für die Verschlüsselung sehr schnell. Allerdings ist die Verwendung diese Zahl nicht sicher für das RSA- Verfahren.

Bei unsere Implementierung haben wir zunächst geschaut ob unsere φ(N) größer ist als 65537 oder ob 65537 kleiner gleich φ(N) ist. Falls es kleiner gleich ist, wird e = 3 gesetzt. Im anderen Fall ist e = 65537. Falls allerdings e nicht teilerfremd zu φ(N) ist , addieren wir e mit 2 und prüfen es erneut, ob e teilerfremd zu φ(N) ist.

2.2.1 Camicael Funktion

Für die Berechnung von φ(N) wird die Eulersche φ- Funktion verwendet. Die φ Funktion gibt die obere Schranke für e an. Es gibt neben der Eulersche φ- Funktion noch eine weitere Funktion, um die obere Schranke für e bestimmt, die Carmicael Funktion, auch λ- Funktion genannt. Diese Funktion berechnet den kleinsten positiven Quotienten der φ- Funktion aus und optimiert somit die obere Grenze. In der unteren Graphik kann

man sehen, dass der Wert von λ(N) ist in den meisten Fällen kleiner ist als φ(N). So kann e möglichst klein gewählt werden. Je kleiner e, desto schneller ist die Verschlüsselung. **[?]**

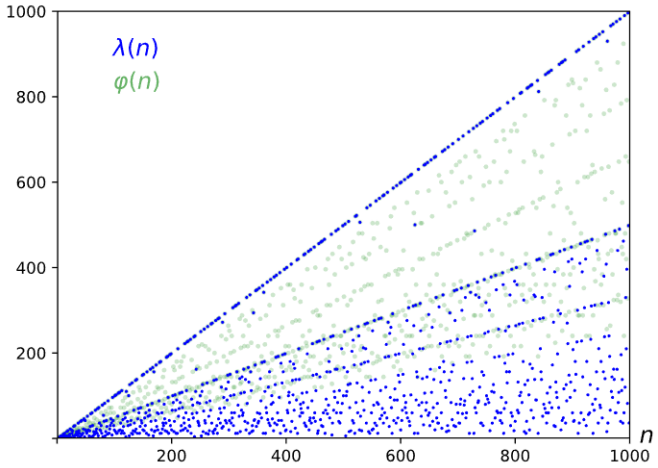

Abbildung 1: Vergleich zwischen φ(N) und λ(N)

Die Formel für die Carmicael- Funktion sieht wie folgt aus:

$$g^m = 1 \mod n \tag{8}$$

Dabei ist n der Eingabewert der Funktion. g sind alle teilerfremden Zahl zu n, welchen in der Funktion berechnet werden. m ist der Rückgabewert. Zur Implementierung der Carmicael Funktion haben wir zunächst die gcd()- Funktion als Hilfsmethode implementiert, welche den größten gemeinsamen Teiler zwischen zwei Zahlen a und b bestimmt. Falls das Ergebnis gleich eins ergibt, bedeutet dies, dass beide Zahlen teilerfremd sind und a kann so als g verwendet werden. Die Implementierung des Algorithmus ist einfach gehalten. Sie beinhaltet einen for- Loop, für die Überprüfung von allen teilerfremden Zahlen die kleiner als n sind und einen while- Loop, der Überprüft bei welchem Exponenten die Eigenschaft von (7) erfüllt wird. Falls eine teilerfremde Zahl gefunden wird, wird nach dem Bestimmen des Exponenten der Rückgabewert k aktualisiert. Dabei wird das kleinste gemeinsame Vielfache zwischen k und dem Exponenten berechnet.

So war die eigentliche Implementierung. Jedoch sind wir bei dieser Art von Implementierung auf ein Problem gestoßen, denn es funktioniert nur für kleine Zahl, in eine Größenordnung von sechs. Deshalb haben wir letztendlich für eine andere Implementierungsvariante der λ Funktion entschieden, mit der folgende Formel:

$$\lambda(N) = \frac{p-1 \cdot q-1}{gcd(p-1, q-1)} \tag{9}$$

Diese Implementierung nimmt die zwei Parameter p und q als Eingabe und berechnet mit Hilfe von der gcd- Funktion den λ Wert. Die andere Implementierung dient ledigtlich nur als Hilfe und wird im richtigen Code nicht verwendet.

2.3 Der erweiterte Euklidische Algorithmus

Bevor wir mit dem erweiterten Euklidischen Algorithmus beginnen, sollten wir zunächst verstehen, was der Euklidische Algorithmus bewirkt. Er ist der effizienteste Weg, um den größten gemeinsamen Teiler GCD zwischen zwei Zahlen zu bestimmen. In unserer Aufgabe müssen wir prüfen, ob die obere Schranke phi(N) und unser Verschlüsselungsexponent e teilerfremd sind. Die grundlegende Idee des Euklidischen Algorithmus ist, dass man den größten gemeinsamen Teiler zwischen zwei Zahlen bestimmt. Das Verfahren ist wie folgt:

1. Die Funktion bekommt zwei Eingaben, a und b. Daraus berechnet man sich r, welcher den Rest von der Division zwischen a und b bildet.
2. Falls r gleich null ist, ist der größte gemeinsame Teiler von a und b gleich b und wir geben ihn als Ausgabe aus.
3. Falls r nicht gleich null ist, ersetzen wir a durch b und b durch r und beginnen erneut bei Schritt 1.

Wie der Name schon verrät, ist der erweiterte Euklidische Algorithmus eine Erweiterung des ursprünglichen Euklidischen Algorithmus. Beide Verfahren berechnen den GCD zwischen zwei ganzen Zahlen. Der erweiterte Algorithmus gibt uns außerdem noch den Koeffizienten der Bezout-Identität an. Es sind zwei ganze Zahlen x und y und stehen wie folgt im Zusammenhang:

$$a \cdot x + b \cdot y = gcd(a, b) \tag{10}$$

Wenn man die zwei Variable a und b hat, kann man x und y mit Hilfe von modulo multiplikative Inverse berechnen.

Die Implementierung vom EEA funktioniert wie folgt. Wir initialisieren zuerst x = 1 und y = 0. Diese sind unsere Koeffizienten der Bezout-Identität. Wir führen den euklidischen Algorithmus aus. Bei jedem Schritt werden außerdem die zwei Koeffizienten x und y mit folgender Formel aktualisiert.

$$x_{new} = y, y_{new} = x - (floor(r/s)) \cdot y \tag{11}$$

Die Variablen r und s sind Reste und Divisor des euklidischen Algorithmus. Die Funktion floor() gibt die größte Zahl, die kleiner als der Eingabewert ist, zurück. Wenn der euklidische Algorithmus terminiert, erhälten man das Ergebnis von GCD und die zwei Koeffizienten der Bezout-Identität, x und y.[**?**] Für die Modulo Inverse von a $\mod$ b kann man mit Hilfe von x $\mod$ b berechnen, falls gcd(a,b) = 1 ist. Im anderen Fall gibt es keine Modulo Inverse.

In unsere Aufgabe sieht die Gleichung folgendermaßen aus:

$$e \cdot d + \varphi(N) \cdot y = gcd(e, \varphi(N)) = 1 \tag{12}$$

Mit Hilfe des EEAs und den bekannten öffentlichen Schlüssel (e, N) wollen wir nun den privaten Schlüssel (d, N) berechnen. Mit einer Umformung ergibt sich:

$$e \cdot d \equiv 1 \mod (\varphi(N)) \tag{13}$$

Lässt sich auch d aus e und φ (N) bestimmen. Wir können den EEA nicht nur für die Berechnung von GCD zwischen e und φ(N) verwenden, sondern auch die Modulo Inverse vom öffentlichen Schlüssel. Wobei das Ergebnis den privaten Schlüssel d ergbit.

2.4 Sicherheit von RSA

Bereits bei dem EEA-Teil haben wir die Formel ed = 1 mod φ(n) vorgestellt. Dabei sei (e, N) ein uns bekannter öffentlicher Schlüssel. Um die Nachricht entschlüsseln zu können, muss man wesentlich nur den privaten Schlüssel d herausfinden.

$$d = e^{-1} \mod \varphi(n) \tag{14}$$

Dabei sei φ(N) wie folgt definiert:

$$\varphi(n) = \varphi(p) \cdot \varphi(q) = (p-1) \cdot (q-1), n = p \cdot q \tag{15}$$

Die Variable e ist uns bereits bekannt. Der schwierige Teil liegt wesentlich daran φ(n) herauszufinden. Mit anderen Worten muss man die zwei Primfaktoren p und q für n finden. Die Primfaktorzerlegung einer sehr großen Zahl zu finden, ist fast unmöglich. In der Praxis würde so ein Algorithmus nicht in polynomieller Zeit laufen. Mit dem „Schroeppel factoring algorithm“ wird die Faktorisierungszeit bei unterschiedlichen n- Längen dargestellt. Bereits bei einer 75 steligen n kann dies zu einer Laufzeit von mehreren Monate führen. Selbst der heutzutage meistverwendete Primfaktorzerlegungsalgorithmus, Trial division, hat eine Laufzeit von O(n^2). Somit ist die Sicherheit vom RSA hauptsächlich auf der Schwierigkeit der Faktorisierung basiert. **[?]**

Länge n	Anzahl Operationen	Zeit
50	1.4×10^{10}	3.9 Stunden
75	9×10^{12}	104 Tage
100	2.3×10^{15}	74 Jahre
200	1.2×10^{23}	3.8×10^{9} Jahre
300	1.5×10^{29}	4.9×10^{15} Jahre

Tabelle 1: Laufzeit Schroeppel factoring

Bei unsere Implementierung ist unser N auf 19 Stellen eingeschränkt. Deshalb sind unsere Schlüssel unsicher und N kann sehr mit schnell mit der Primfaktorzerlegungsalgorithmus in zwei Primzahlen zerlegt werden.

Ein weiterer Punkt wovon die Sicherheit von RSA auch abhängig ist, ist die Wahl von e. In Sektion 2.3 wurde erwähnt, dass die kleinste Zahl für e die Drei ist. Auch

wenn der Verschlüsselungsexponent bei keinere Zahlen sehr schnell verläuft, ist die Sicherheit nicht vielversprechend. Im Fall, dass m^e kleiner als N ist und auch wenn die Verschlüsselung der Nachricht sehr schnell verläfut, bedeutet es mit großen Wahrscheinlichkeit, dass der Verschlüsselungsexponent eine kleine Zahl, meistens die Drei ist. Solcher Angriff wird auch als Timing Attack genannt. Der Angreifer benutzt die Zeit für die Verschlüsselung der Nachricht, um Informationen über den Verschlüsselungsprozess zugewinnen. [**?**]

Das Verfahren was wir implementiert haben wird auch Textbook RSA genannt. Es ist RSA ohne Padding. Das Verfahren verwendet viele mathematische Operationen und kann zu manchen Schwächen führen. Um diese zu vermeiden wird das Padding für RSA hinzugefügt. Ein sehr häufig verwendetes Padding Verfahren für RSA ist das öptimal asymmetric encryption padding". [**?**]

3 Korrektheit/Genauigkeit

4 Performanzanalyse

Um die Performanz unserer Lösungsansätze und Optimierungen zu analysieren haben wir insgesamt sechs verschiedene Varianten der RSA in C und Assembly programmiert, bei denen paarweise einige Parameter unterschiedlich sind. Für ein besseres Verständis nennen wir zuerst die Variablen und nutzen ab jetzt VX für die X-te Variante:

V0: Die Assembly-Implementierung, aka die Hauptimlementierung, mit dem linearen Kongruenzgenerator für die Zufallszahlgenerierung. V1: Die C-Implementierung mit dem kryptographischen Zufallszahlgenerator arc4random. V2: Die C-Implementierung mit dem linearen Kongruenzgenerator für die Zufallszahlgenerierung. Es ist sozusagen eine C Übersetzung der V0. Der kleine Unterschied wird unten erklärt. V3: Eine Variante von V2. Wir haben nur bei der Primecheck-Algorithmus die sqrt(n)-Funktion von C benutzt und die Laufzeitdifferenz verglichen. (Detaillierte Erklärung ist beim Teil 2.1 unter Primecheck-Algorithmus zu finden.) V4: Eine Variante von V2. Anstatt phi-Funktion wird die lambda-Funktion benutzt. (Detaillierte Erklärung ist beim Teil 2.2.1 zu finden.) V5: Eine Variante von V0. Es wird hier für Vergleichszwecke keine Zufallszahlgenerator verwendet und die p und q wurden aus der Zahlen 0xfffffffe und 0xfffffffd berechnet. Detaillierte Erklärung finden Sie unten. V6: Eine Variante von V2. Gleiche Modifizierungen wie bei V5.

Für das Messen der Werte haben wir die clock_gettime()-Funktion mit der Uhr CLOCK_MONOTONIC verwendet. Alle Varianten wurden auf einem Raspberry Pi 3 mit Broadcom BCM2837 Prozessor mit 4 x ARM Cortex-A53, 1.2 GHz, 1 GB Arbeitsspeicher, Debian 10 (Buster), AArch64, Linux-Kernel 4.8.16, gcc 8.3.0 und Optimierungsstufe -O3 ausgeführt.

Obwohl V0 die Hauptimplementierung ist, sind die Laufzeitergebnisse davon wertlos. Da RSA auf Zufallszahlen basiert, hängt die Laufzeit der V0 stark davon ab, wie viel es dauert, die passenden p und q zu finden. Trotz der Nutzung des Pseudo-Zufallszahlgenerators, sind die daraus entstehende Werte, mindestens für uns, nicht

vorrechenbar. Die Grund dafür ist die Nutzung des Werts im Speicher [0x12345678] als Seed. Der unbekannte Wert weicht den Pseudo-Zufallsgenerator von Pseudo-sein ab. Bei C-Implementierungen ist das nicht der Fall, weil wir explizit einen unsigned long als Seed angeben, anstatt eine Speicheradresse. Damit die Vergleichung fairer ist, haben wir uns aber dafür entschieden, die gleiche Methode wie bei V5 in V2 zu nutzen. Das haben wir V6 genannt. Also, deshalb untersuchen wir die Laufzeiten von V5 und V6 als Referenz zum V0 und V2.

Länge n	Anzahl Operationen	Zeit
50	1.4×10^{10}	3.9 Stunden
75	9×10^{12}	104 Tage
100	2.3×10^{15}	74 Jahre
200	1.2×10^{23}	3.8×10^{9} Jahre
300	1.5×10^{29}	4.9×10^{15} Jahre

Tabelle 2: Laufzeit Schroeppel factoring

Bei V5 und V6 sind als "Zufallszahlen" zwei unidentische große zusammengesetzte Zahlen gewählt. So ist auch der Teil mit von nicht zusammengesetzten Zahlen eine Primzahl zu erstellen mitgerechnet. Die Ergebnisse sollen suboptimale Fälle begehen, wenn nicht die Schlimmsten. Man merkt auf der untenstehenden Tabelle, dass der gcc -O3 eine bessere Arbeit als uns geleistet hat. Die Werte sind aber sehr nah aneinander mit mit Referenz zum V5.

Wie in 2.1 erwähnt, die Praktikumsordnung verbietet uns das Nutzen von komplexen arithmetischen Operationen wie sqrt(n). Deshalb haben wir den erklärten Umweg gefunden und benutzt. Aus unserem Sicht ist aber ein Vergleich nötig, um herauszufinden, ob zwischen den zwei Wege einen Unterschied beim Laufzeit gibt. Die untere Tabelle zeigt die Laufzeiten der Varianten V2 und V3.

Länge n	Anzahl Operationen	Zeit
50	1.4×10^{10}	3.9 Stunden
75	9×10^{12}	104 Tage
100	2.3×10^{15}	74 Jahre
200	1.2×10^{23}	3.8×10^{9} Jahre
300	1.5×10^{29}	4.9×10^{15} Jahre

Tabelle 3: Laufzeit Schroeppel factoring

Obwohl es in der Aufgabenstellung nicht steht, haben wir zusätzlich einen Ver- und Entschlüsseler programmiert. Unten auf der Tabelle sieht man, wie das Nutzen von Carmichaels Lambda-Funktion, wie in 2.2.1 erklärt, die Laufzeit beeinflusst. In V2 verwendet man die Euler'sche phi-Funktion und in V4 die lambda-Funktion.

5 Zusammenfassung und Ausblick

In dieser Projektarbeit hatten wir die Aufgabe mit dem Verschlüsselungsverfahren RSA, welches in der heutigen Zeit immer noch oft verwendet wird, zu beschäftigen. Dabei war unsere genaue Aufgabe die Schlüsselgenerierung von RSA in Assembly zu Implemetieren. Neben den ganzen Implementierungen sollten wir außerdem noch ein Rahmenprogramm in C schreiben. Das Rahmenprogramm sollte unterschiedliche Eingabenoptimierungen annehmen, sowie eine Hilfestellung für die Funktionsweise des Programmes. Auch eine Performanzanalyse haben wir durchgeführen müssen. Mit Hilfe davon konnte man festestellen feststellen wie die Laufzeit mit Hilfe vom Assembly Code optimiert werden.

Für die Implementierung der Schlüsselgenerierung haben wir zunächst das Problem in drei Teile aufgeteilt. Wir haben mit der Variable N angefangen. Dabei sei die schwierigkeit zwei zufällige Primezahl zu generieren. Anschließend kommt die Wahl von e in Frage. Dafür haben wir die zwei häufigetsen gewählten Zahl für e rausgesucht und bei bestimmten Fällen einer dieser Zahlen verwendet. Zum Schluss musst noch festgelegt werden, welche Zahl d ist. Mit dem Verfahren von EEA kann man aus e und N die Variable d herleiten.

Die generierung von den drei Variablen verläuft nach strickten mathematischen Voraussetzungen und deshalb sind die immer genau.

6 Bildequellen

Abbilldung 1: `https://en.wikipedia.org/wiki/Carmichael_function#/media/File:CarmichaelLambda.svg`(03.02.2023)