Bedeutung von starken Primzahlen für die heutige Zeit

Facharbeit

Im Fach

Mathematik

eingereicht an dem

Carl-Friedrich-Gauß Gymnasium

von

Winterer, Mathis Aaron

Betreuer der Seminararbeit: E. Menzel

Frankfurt an der Oder, 21. November 2024

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

Kryptographie und Verschlüsselungssysteme wurden schon im Antiken Rom verwendet um Nachrichten zu verschlüsseln[Aic22], doch heutzutage wären Verschlüsselungssysteme wie der „Cäsar-Chiffre“ gegenüber eines Computers nutzlos. Die Rolle von Kryptographie ist aufgrund des leichten Zugangs der Allgemeinheit zu leistungsstarken Computern, welche das Brechen von „schwachen“ Verschlüsselungssystemen automatisiert und beschleunigt haben, um einiges angestiegen. Aufgrund der nun verfügbaren Rechenleistung wurde die Notwendigkeit für stärkere Verschlüsselungssysteme immer größer. Zur Zeit verwendete Verschlüsselungssysteme sind so konzipiert, dass sie mathematisch schlecht rückwärts zu berechnen sind. Hierzu werden standardmäßig große Primzahlen verwendet, da zur Zeit kein effizienter Algorithmus zur Berechnung von großen Primzahlfaktoren bekannt ist, welcher nicht die Verwendung eines Quantencomputers erfordern würde. Das „brute-forcing“ moderner Verschlüsselungssysteme ist so ineffizient, dass nur ein Teilschritt bis zu fünfzehnhundert Jahre dauern kann[Kle+10]. „Für unsere Berechnungen waren mehr als 10^{20} Operationen erforderlich. Mit dem Äquivalent von fast 2000 Jahren Rechenzeit auf einem AMD Opteron mit einem Kern von 2,2 GHz […]“[Kle+10]. Verschlüsselungssysteme sind aufgrund dieser hohen Anforderungen gegenüber der Resistenz gegen Angriffe äußerst komplexe Systeme welche in der Informationssicherheit eine entscheidende Rolle bei der Sicherung von sensiblen Daten sowohl in der Übertragung als auch bei der Speicherung spielen[Aic22]. Das Ziel dieser Facharbeit ist es den weit verbreiteten und etablierten RSA-Algorithmus zu Analysieren und zu Implementieren. Hierfür werden die einzelnen Teilschritte und verwendeten mathematischen Prinzipien und Funktionen erläutert und beispielhaft dargestellt sowie als Teil eines Programms in C# implementiert, so dass die Funktionen der Schlüsselerzeugung, der Verschlüsselung, der Entschlüsselung und der Signierung verfügbar sind.

1.1 Hintergrund der RSA-Verschlüsselung

Entwickelt von : …

1.2 Anwendungsbereiche der RSA-Verschlüsselung

Benutzt für : …

2. Mathematische Grundlagen der RSA-Verschlüsselung

2.1 Kleiner fermatscher Satz

Der kleine fermatsche Satz, benannt nach seinem Entdecker Pierre de Fermat, ist ein Satz in der Zahlentheorie, welcher eine Kongruenz zwischen einer natürlichen Zahl a und einer Primzahl p herstellt.[Wol]
Wenn a eine beliebige natürliche Zahl, die nicht durch p teilbar ist, und p eine beliebige Primzahl seien, dann gilt:

$$\begin{aligned} &\{p \in \mathbb{P}\} \\ &\{a \in \mathbb{N} \mid ggT(a,p) = 1\} \\ &a^p \equiv a \pmod{p} \end{aligned} \tag{2.1}$$

Dies bedeutet, dass der Rest bei der Division von a^p durch p immer a ist:

$$\begin{aligned} &2^3 \equiv 2 \pmod{3} \\ &4^{17} \equiv 4 \pmod{17} \\ &5^7 \equiv 5 \pmod{7} \end{aligned} \tag{2.2}$$

Es finden sich in verschiedenen Quellen[Wol][Weic] auch leicht abgeänderte Formeln wie:

$$\begin{aligned} &(a^{p-1} - 1) \bmod p = 0 \\ &a^{p-1} \equiv 1 \pmod{p} \end{aligned} \tag{2.3}$$

Für meine Zwecke verwende ich die als drittes beschriebene $a^{p-1} \equiv 1 \pmod{p}$ Formel, da die erste und zweite Formel keinen Mehrwert bei meiner Anwendung bieten.

2.2 Primzahltests

2.2.1 Miller-Rabin-Test

Der Miller-Rabin-Test ist ein probalistischer Primzahltest, dies bedeutet, dass der Algorithmus nicht immer zweifelsfrei korrekte Ergebnisse berechnet, jedoch kann die Genauigkeit der Ergenisse durch mehrfaches Durchlaufen des Algorithmus verbessert werden. Der Miller-Rabin-Test hat für eine Iteration eine Wahrscheinlichtkeit kleiner als $\frac{1}{4}$ eine zusammengesetze Zahl als Primzahl zu erkennen. Aufgrund dessen ist es empfehlenswert den Algorithmus über mehrere Iterationen auszuführen, sodass die Wahrscheinlichtkeit des Fehlers vernachlässigbar wird. Nach i Iterationen ist die Wahrscheinlichtkeit des Fehlers $\leq (\frac{1}{4})^i$. Somit ist sie für 10 Iterationen schon rund 0.00000095367431640625 oder geringer, was für die Primzahlselektion des RSA-Algorithmus ausreichend ist. Abbildung 2.2.1 zeigt den Verlauf der Fehlerwahrscheinlichkeit von 1 bis 100 Iterationen und illustriert deutlich, dass der Miller-Rabin-Test schon mit wenigen Iterationen einen ausreichend präziser Primzahltest bietet.[Rab80]

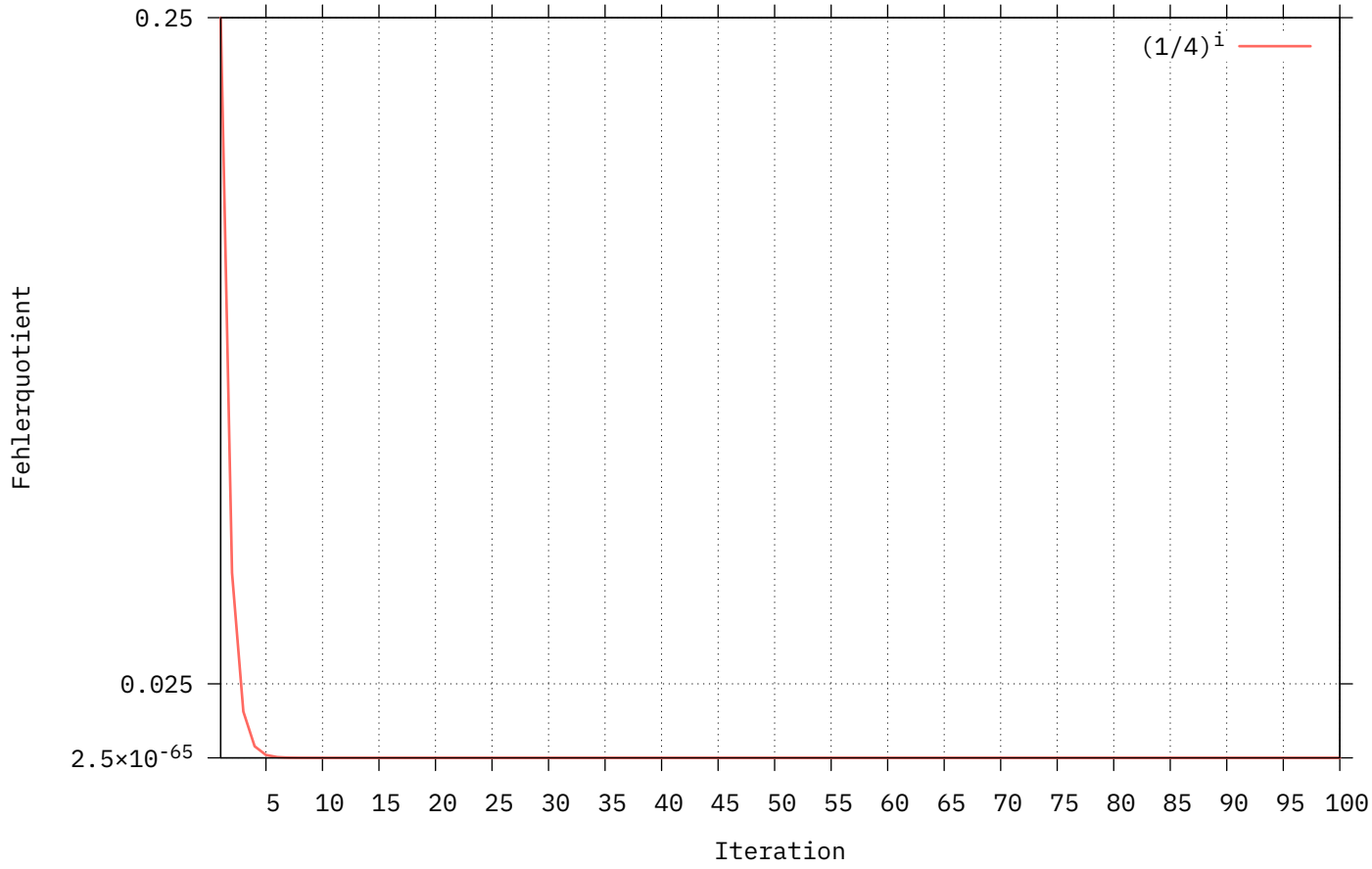

Abbildung 2.1: Fehlerquotient für $\{i \in \mathbb{N} \mid 1 \leq i \leq 100\}$

2.2.2 Fermatscher Primzahltest

Der kleine fermatsche Satz kann als rudimentärer Primzahltest verwendet werden, da die Kongruenz zwischen a und p nur gegeben ist, wenn p eine Primzahl ist. Somit kann durch das iterative Testen dieser Kongruenz mit verschiedenen Basen a ein Schluss auf die Primalität von p gezogen werden. Die folgende Abbildung 2.2 implementiert eine Funktion, welche zurückgibt ob n wahrscheinlich prim oder definitiv zusammengesetzt ist. Der Parameter k erlaubt es die Präzision der Funktion durch weitere Durchläufe zu verbessern.

```
private static bool Fermat(BigInteger n, int k)
{
  // Falls die zu überprüfende Zahl 3 oder 2 ist
  if (n == 3 || n == 2)
    // Wahr zurückgeben
    return true;
  // Führe den Primzahltest für k Iterationen aus, beziehungsweise bis er fehlschlägt
  for (int _ = 0; _ < k; _++)
  {
    // Label zu dem gesprungen wird falls keine teilerfremde Base a gefunden werden konnte
    Restart:
    // Zufällige Base a mit 2 ≤ a ≤ n-2
    BigInteger a = GetRandomBigIntInRange(n.GetByteCount(), 2, n - 2);
    // Zählt die Anzahl der Versuche eine teilerfremde Base a basierend auf der selben Zufallszahl zu finden
    int coprimeRuns = 0;
    // Label zu dem gesprungen wird falls die derzeitige Base a nicht teilerfremd ist
    NotCoprime:
    // Falls die Base a mehr als 5 mal nicht teilerfremd war wird eine neue Zufallszahl generiert
    if (coprimeRuns > 5)
      goto Restart;
    // Untersuche ob a und n teilerfremd sind
    if (BigInteger.GreatestCommonDivisor(a, n) != 1)
    {
      // erhöhe die Base a um eins
      a++;
      // erhöhe die Anzahl der Versuche um eins
      coprimeRuns++;
      // springe zum vorherigen Label
      goto NotCoprime;
    }
    // Berechne a^n-1 mod n
    BigInteger res = ModPow(a, n - 1, n);
    // Falls der Rest nicht eins ist ist n nicht prim
    if (res != 1)
    {
      // Gebe falsch zurück
      return false;
    }
  }
}
```

Abbildung 2.2: Implementation eines Primzahltestes basierend auf dem kleinen fermatschen Satz

2.3 Eulersche Phi-Funktion

Die Eulersche Phi-Funktion, auch bekannt als eulersche Totientfunktion, ist eine von Leonhard Euler 1763 formulierte Funktion der Zahlentheorie, welche die Summe der teilerfremden Zahlen zu n darstellt[Eul]:

$$\begin{aligned} & n \in \mathbb{N}^+ \\ & \{\varphi(n) \in \mathbb{N}^+ \mid 0 \leq \varphi(n) \leq n-1\} \\ & \varphi(n) = n \times \prod_{p|n} \left(1 - \frac{1}{p}\right) \end{aligned} \tag{2.4}$$

$\varphi(n)$ ist der Totient von n.

$$\varphi(5) = 4 \qquad \varphi(5) = n-1, \text{ da n prim ist} \tag{2.5}$$

$$\varphi(9) = 6 \qquad \text{teilerfremd zu } 1_1,\ 2_2,\ 4_3,\ 5_4,\ 7_5,\ 8_6 \tag{2.6}$$

$$\varphi(8) = 4 \qquad \text{teilerfremd zu } 1_1,\ 3_2,\ 5_3,\ 7_4 \tag{2.7}$$

$$\varphi(16) = 8 \qquad \text{teilerfremd zu } 1_1,\ 3_2,\ 5_3,\ 7_4,\ 9_5,\ 11_6,\ 13_7,\ 15_8 \tag{2.8}$$

In Gleichungen (2.5), (2.7) und (2.8) sind zwei Spezialfälle für Werte n der eulerschen Phi-Funktion zu sehen; Primzahlen und Potenzen zur Basis zwei:

1. Für $n \in \mathbb{P}$ ist $\varphi(n)$ immer gleich $n-1$.[Weie]
2. Für $n = 2^k \wedge k \geq 1$ ist $\varphi(2^k) = 2^{k-1}$.

Weitere Attribute der Funktion und ihrer Parameter sind:

1. Für $n \geq 3$ ist $\varphi(n)$ immer gerade, sprich $\varphi(n) = 2k$.[Weie]
2. Konventionell ist $\varphi(0) = 1$ definiert.[Weie]

Die eulersche Phi-Funktion findet auch Anwendung in einer Verallgemeinerung des kleinen fermatschen Satzes, welche von Euler auch 1763 publiziert wurde:

$$a^{\varphi(n)} \equiv 1 \pmod{()n} \tag{2.9}$$

Für alle Zahlen a welche teilerfremd zu n sind.[Weib]
2.3 zeigt die Verteilung von $\varphi(n)$ für die ersten eintausend Werte n.

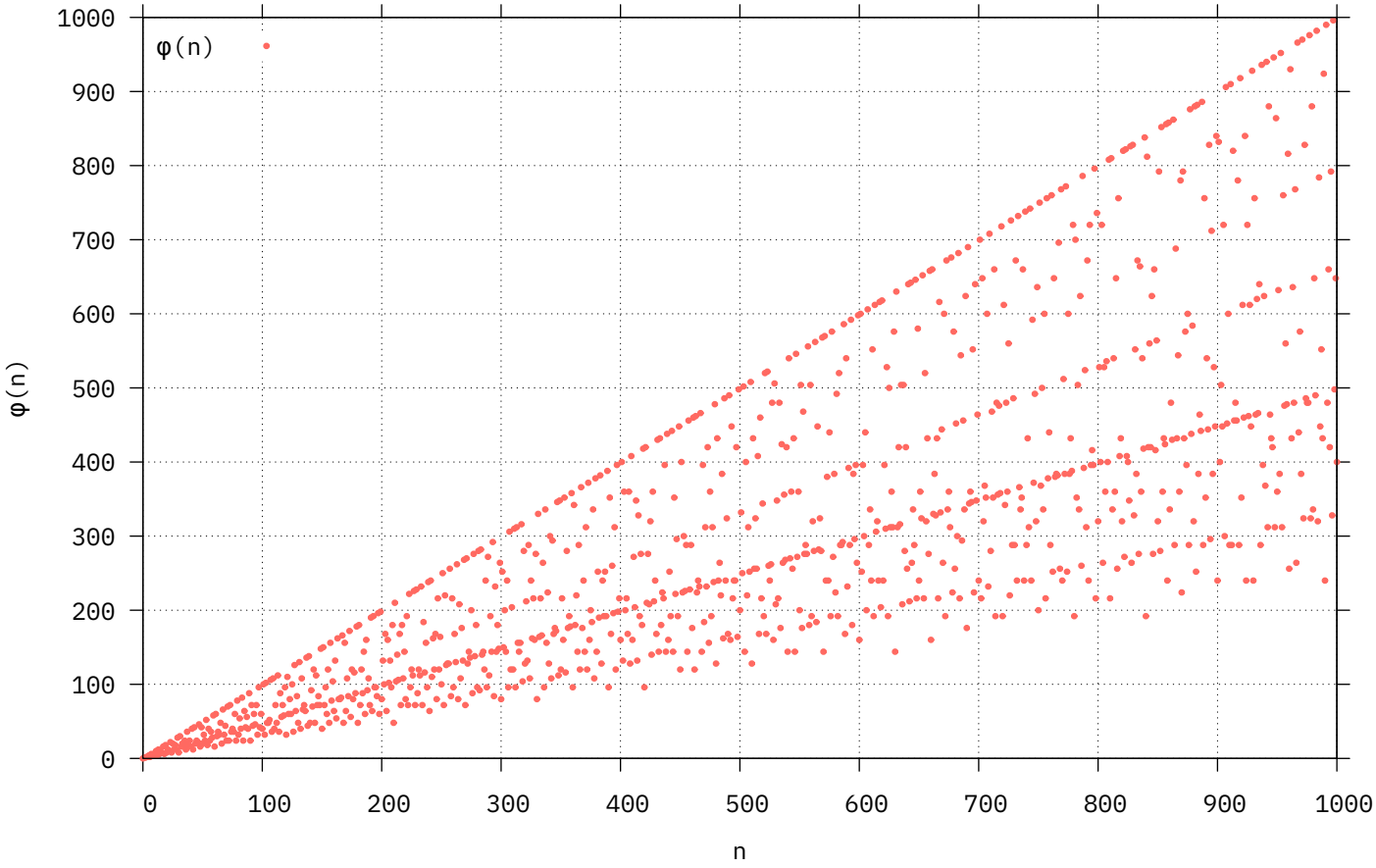

Abbildung 2.3: Eulersche Phi-Funktion $\varphi(n)$ für $\{n \in \mathbb{N} \mid 0 \leq n \leq 1000\}$

2.4 Carmichael-Funktion

Die Carmichael-Funktion $\lambda(n)$ gibt für eine natürliche Zahl n die kleinste positive ganze Zahl an, sodass für alle zu n teilerfremden Zahlen k gilt:

$$k^{\lambda(n)} \equiv 1 \pmod{n} \tag{2.10}$$

Die Funktion kann wie folgt definiert werden:

$$\lambda(n) = \text{kgV}\left[(p_i - 1)p_i^{a_i - 1}\right]_i \tag{2.11}$$

[Weia]
Hierbei steht p_i für den i-ten Primfaktor von n und a_i für den Exponent des i-ten Primfaktores.[Weid]
Um $\lambda(n)$ einer Zahl zu berechnen werden folgende Schritte verwendet:

1. Zahl in Primfaktoren zerlegen
2. Alle Primfaktoren in $(p_i - 1)p_i^{a_i - 1}$ einsetzen und die Ergebnisse speichern
3. den kleinsten gemeinsamen Teiler aus allen Ergebnissen bilden

Beispielhalft sei n gleich 15:

$$\begin{aligned}
n &= 15 = 3^1 \cdot 5^1 && \text{Darstellung durch Primfaktoren} \\
p_0 &= 3 && \text{Erster Primfaktor von } 15 \\
p_1 &= 5 && \text{Zweiter Primfaktor von } 15 \\
a_0 &= 1 && \text{Exponent des ersten Primfaktors } 3 \\
a_1 &= 1 && \text{Exponent des zweiten Primfaktors } 15 \\
k &= \left((p_0 - 1) \cdot p_0^{a_0 - 1}; (p_1 - 1) \cdot p_1^{a_1 - 1}\right) && \text{Folge } k \text{ welche Ergebnisse enthält} \\
\hookrightarrow &= \ ((1; 12)) \\
\lambda(15) &= \text{kgV}(k) = 4
\end{aligned} \tag{2.12}$$

Oder für $n \in \mathbb{P}$:

$$\begin{aligned}
n &= 17 = 17^1 && \text{Nur 1 und 17 als Primfaktor, da prim} \\
p_0 &= 17 \\
a_0 &= 1 \\
k &= ((17 - 1) \cdot 17^{1-1}) \\
\lambda(17) &= \text{kgV}(k) = 16
\end{aligned} \tag{2.13}$$

die Berechnung in 2.13 zeigt einen Sonderfall der Carmichael-Funktion, denn für alle $n \in \mathbb{P}$ ist $\lambda(n) = n - 1$, wie in 2.14 weiter zu sehen ist.

$$
\begin{aligned}
\lambda(2) &= 1 \\
\lambda(3) &= 2 \\
\lambda(5) &= 4 \\
\lambda(7) &= 6 \\
\lambda(11) &= 10 \\
\lambda(13) &= 12 \\
&\dots
\end{aligned}
\tag{2.14}
$$

Abbildung 2.4 zeigt die Verteilung von $\lambda(n)$ für die ersten eintausend Werte n. Abbildung 2.5 zeigt eine kombinierte Verteilung von sowohl der eulerschen Phi-Funktion als auch der Carmichael-Funktion.

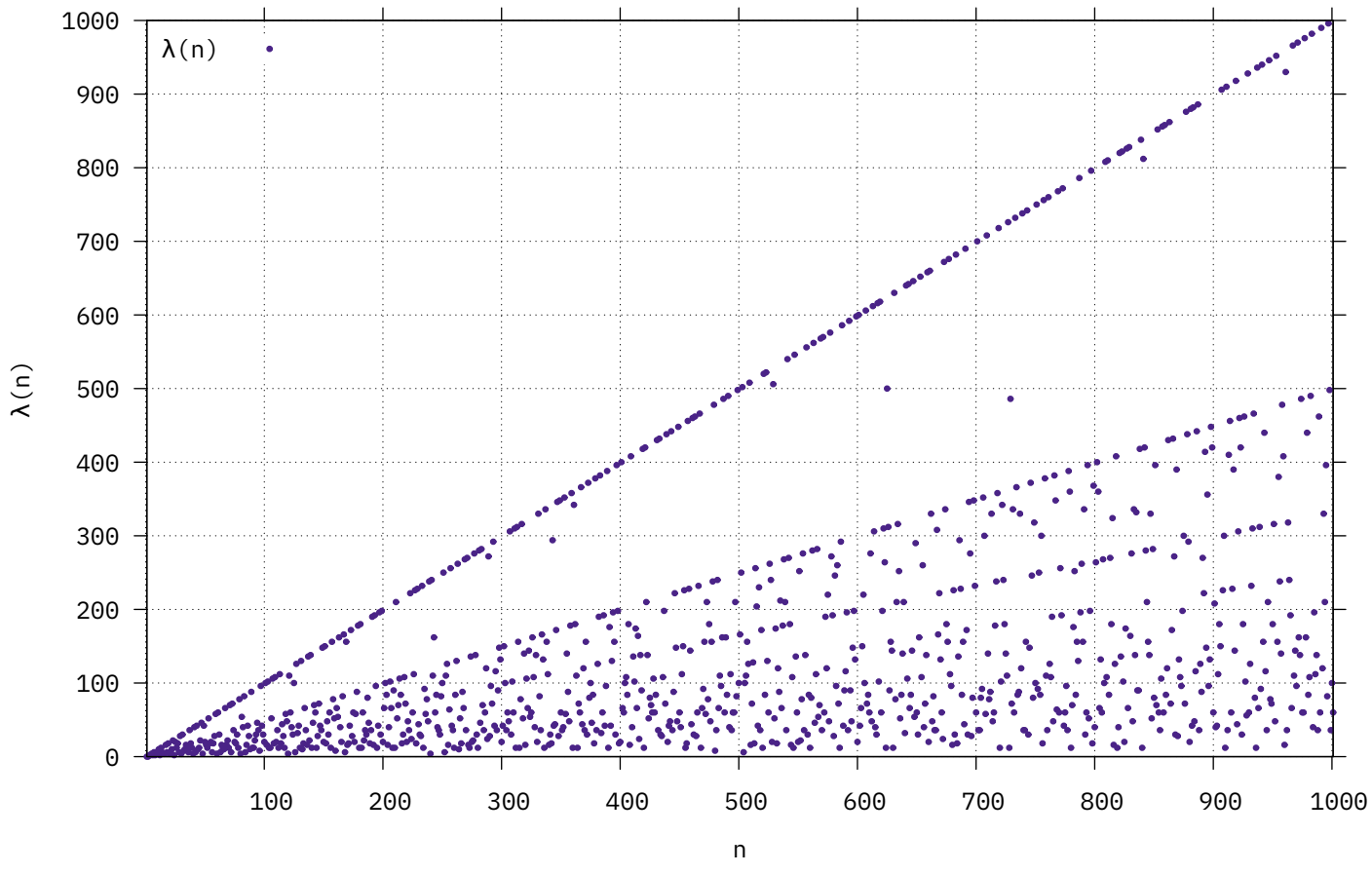

Abbildung 2.4: Carmichael-Funktion $\lambda(n)$ für $\{n \in \mathbb{N} \mid 1 \leq n \leq 1001\}$

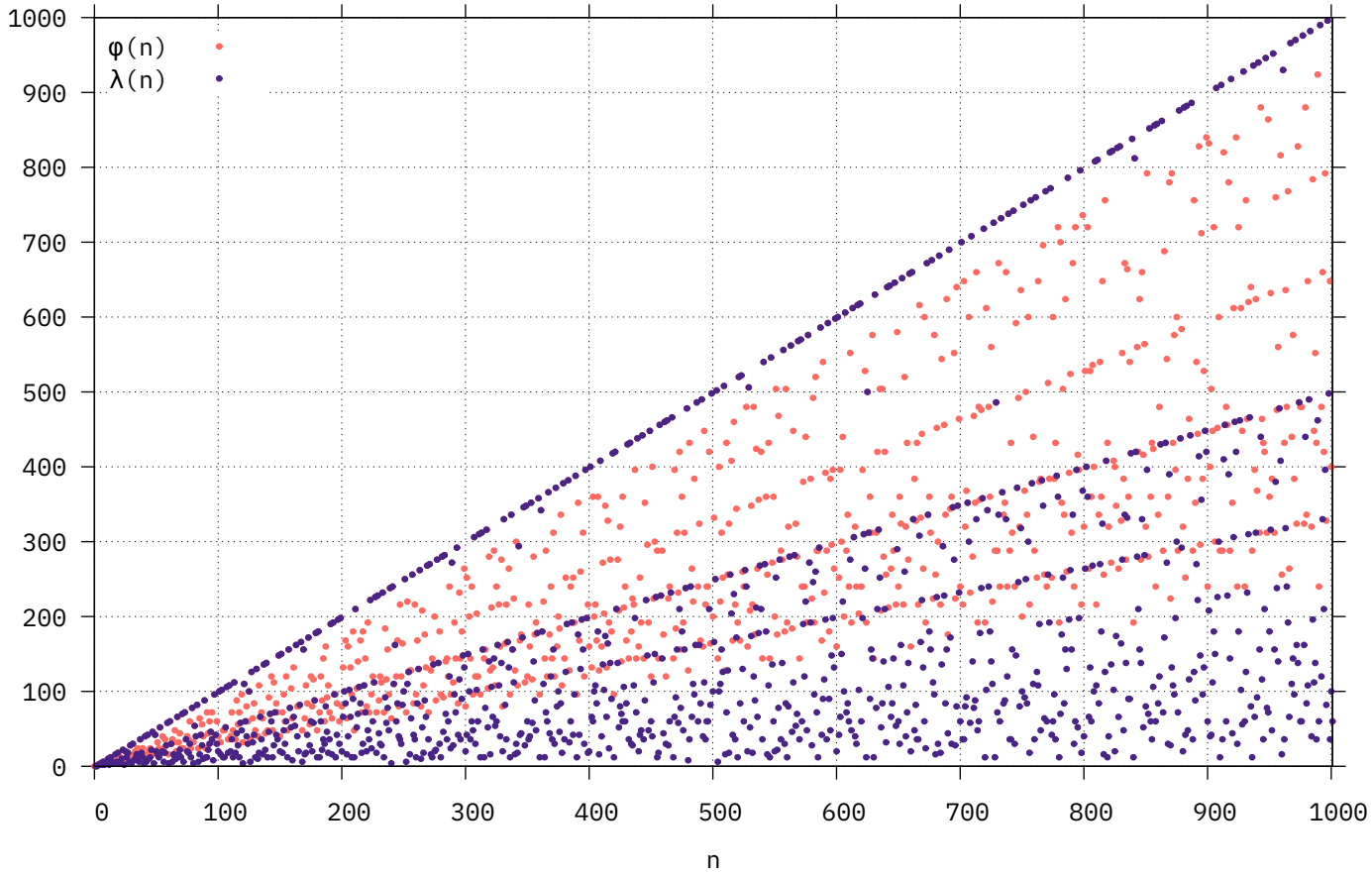

Abbildung 2.5: Carmichael-Funktion $\lambda(n)$ für $\{n \in \mathbb{N} \mid 1 \leq n \leq 1001\}$ und eulersche Phi-Funktion $\varphi(n)$ für $\{n \in \mathbb{N} \mid 0 \leq n \leq 1000\}$

2.5 Euklidischer Algorithmus

A.

2.5.1 Erweiterter Euklidischer Algorithmus

2.6 Chinesischer Restsatz

3. Anwendungsbeispiel

3.1 Voraussetzungen

Für die Erzeugung eines Schlüsselpaares werden zwei zufällige große Primzahlen p und q so gewählt, dass eine Fastprimzahl $n = pq$ berechnet werden kann. Im Verlauf dieser Arbeit wird immer angenommen, dass p und q die selbe Bitgröße besitzen $pbit = qbit$, wobei dies für den RSA-Algorithmus nicht zwingend notwendig ist. Um das Errechnen von p und q durch Faktorisierung von n zu Erschweren sollten Primzahlen für p und q gewählt werden, welche beide „groß“[1] und „weit“ auseinander liegen. Beide Primfaktoren p und q sind durch die Bitgröße von n, für welche in der Regel eine Zweierpotenz gewählt wird, beschränkt, wobei $pbit$ und $qbit$ in der Regel entweder gleich groß oder annähernd gleich groß sind. $pbit$ und $qbit$ seien für jeden Wert p und q als die Bitgröße dieser definiert. Für $p \geq 2$ sei $q \geq 3 \land q \neq 2$ und für $p \geq 3$ sei $q \geq 2 \land q \neq 3$, da dies die kleinsten Primzahlen sind für welche ein n berechnet werden kann:

$$
\begin{aligned}
&\{ p \in \mathbb{P} \mid (q = 2 \implies 3 \leq p) \oplus (q = 3 \implies (2 \leq p \land p \neq 3)) \} \\
&\{ q \in \mathbb{P} \mid (p = 2 \implies 3 \leq q) \oplus (p = 3 \implies (2 \leq q \land q \neq 3)) \} \\
&\{ pbit \in \mathbb{N} \mid 2 \leq pbit \} \\
&\{ qbit \in \mathbb{N} \mid 2 \leq qbit \} \\
&pbit = \left\lfloor \frac{\ln p}{\ln 2} \right\rfloor + 1 = \lfloor \log_2 p \rfloor + 1 \\
&qbit = \left\lfloor \frac{\ln q}{\ln 2} \right\rfloor + 1 = \lfloor \log_2 q \rfloor + 1
\end{aligned}
\tag{3.1}
$$

Somit sei für eine Zahl $p \coloneqq 2^{2048} - 1$ die Bitgröße $pbit$ dementsprechend:

$$
pbit = \lfloor \log_2 p \rfloor + 1 = 2048 \tag{3.2}
$$

Folgend sei für jeden Wert n die Bitgröße $nbit$:

$$
\begin{aligned}
&\{ nbit \in \mathbb{N} \mid 3 \leq nbit \} \\
&nbit = \left\lfloor \frac{\ln n}{\ln 2} \right\rfloor + 1 = \lfloor \log_2 n \rfloor + 1 \\
&nbit = pbit + qbit
\end{aligned}
\tag{3.3}
$$

Somit ergibt sich für $nbit \coloneqq 4096$, $pbit \coloneqq 2048$ und $qbit \coloneqq 2048$:

$$
\left\{ n \in \mathbb{P} \times \mathbb{P} \mid n \neq 4 \land 6 \leq n \leq 2^{4096} - 1 \right\} \tag{3.4}
$$

[1] Derzeitig sichere RSA-Schlüssel besitzen normalerweise zwischen 1024- und 4096-bit

Zwischen 1991 und 2007 veröffentlichte RSA Laboratories die sogenannten „RSA-Zahlen“. Diese Zahlen waren verschiedene Werte n zwischen 100- und 2048-bit. Ziel dieser Aktion war es die Forschung im Bereich der effizienten Faktorisierung voranzutreiben, dies wurde durch beträchtliche Preisgelder für einige dieser Zahlen bezweckt. Zum derzeitigem Standpunkt ist „RSA-250“ die zuletzt Faktorisierte „RSA-Zahl“ mit 829-bit[Zim20]. Die nächste „RSA-Zahl“ „RSA-260“ enthält 862-bit und wurde bis zu diesem Zeitpunkt nicht faktorisiert. Somit ergeben sich für p, q, n und $nbit$ durch die Primfaktoren von „RSA-250“ neue Mindestanforderungen:

$$\begin{aligned}
&a = 641352894770715802787901901705773890848250147429434472081168596\\
&\hookrightarrow\quad 320245323446302386235987526683477087376619255856946397988533 67\\
&b = 333720275949781565562260106053551142279407603447675546667845209\\
&\hookrightarrow\quad 870238417292100370802574486732968818775657189862580369320627 11\\
&\{p \in \mathbb{P} \mid (p = a \implies a \leq p \wedge b \leq q) \oplus (p = b \implies b \leq p \wedge a \leq q)\}\\
&\{q \in \mathbb{P} \mid (q = a \implies a \leq q \wedge b \leq p) \oplus (q = b \implies b \leq q \wedge a \leq p)\}\\
&\{n \in \mathbb{P} \times \mathbb{P} \mid 2^{862} - 1 < n\}\\
&\{nbit \in \mathbb{N} \mid 862 < nbit\}
\end{aligned} \tag{3.5}$$

Das Bundesamt für Sicherheit in der Informationstechnik empfiehlt die Verwendung von RSA-Schlüsseln mit mehr als 3000-bit, um sicherzustellen, dass der Modulus n derzeit nicht faktorisiert werden kann.[Bun24] Somit ergeben sich erneut neue Definitionen für p, q, n, $pbit$, $qbit$ und $nbit$:

$$\begin{aligned}
&\{p \in \mathbb{P} \mid 2^{1500} - 1 \leq p\}\\
&\{q \in \mathbb{P} \mid 2^{1500} - 1 \leq q\}\\
&\{n \in \mathbb{P} \times \mathbb{P} \mid 2^{3000} - 1 \leq n\}\\
&pbit = \left\lfloor \frac{\ln p}{\ln 2} \right\rfloor + 1 = \lfloor \log_2 p \rfloor + 1 = 1500\\
&qbit = \left\lfloor \frac{\ln q}{\ln 2} \right\rfloor + 1 = \lfloor \log_2 q \rfloor + 1 = 1500\\
&nbit = \left\lfloor \frac{\ln n}{\ln 2} \right\rfloor + 1 = \lfloor \log_2 n \rfloor + 1 = 3000\\
&nbit = pbit + qbit\\
&n = p \cdot q
\end{aligned} \tag{3.6}$$

Als Teil meiner Arbeit werde ich $nbit = 4096$ als Bitgröße verwenden, da 4096-Bit die derzeit größte weit verbreitete Schlüsselgröße bei RSA-Schlüsseln ist. Im Weiteren werde ich davon ausgehen, dass p und q die selbe Bitröße besitzen. Somit ergeben sich für diese Arbeit folgende finale Definitionen für p, q, n, $pbit$, $qbit$ und $nbit$:

$$
\begin{aligned}
&\left\{ p \in \mathbb{P} \mid p \leq 2^{2048} - 1 \right\} \\
&\left\{ q \in \mathbb{P} \mid p \leq 2^{2048} - 1 \right\} \\
&\left\{ p \in \mathbb{P} \mid (p = 2 \implies (2 \leq p \leq 2^{2048} - 1 \land p \neq 3) \land (3 \leq q \leq 2^{2048} - 1)) \oplus \right. \\
&\hookrightarrow \left. (p = 3 \implies (3 \leq p \leq 2^{2048} - 1) \land (2 \leq q \leq 2^{2048} - 1 \land q \neq 3)) \right\} \\
&\left\{ q \in \mathbb{P} \mid (q = 2 \implies (2 \leq q \leq 2^{2048} - 1 \land q \neq 3) \land (3 \leq p \leq 2^{2048} - 1) \oplus \right. \\
&\hookrightarrow \left. (q = 3 \implies (3 \leq q \leq 2^{2048} - 1) \land (2 \leq p \leq 2^{2048} - 1 \land p \neq 3)) \right\} \\
&\left\{ n \in \mathbb{P} \times \mathbb{P} \mid n \leq 2^{4096} - 1 \right\} \\
&pbit = 2048 \\
&qbit = 2048 \\
&nbit = 4096
\end{aligned}
\tag{3.7}
$$

3.2 Schlüsselerzeugung

Die Erzeugung eines Schlüsselpaares basiert auf der Selektion zweier kryptographisch sicheren zufälligen Primzahlen, standardmäßig als p und q bezeichnet, welche meistens beide die gleiche Bitgröße besitzen, dies ist jedoch nicht zwingend notwendig. Die größte Herausforderung stellt das effiziente[2] Finden dieser großen Primzahlen – 1024-bit, 2048-bit, … – dar. Die von mir verwendete Selektion der Primzahlen basiert auf diesem grundlegenden Prozess:

1. Zufällige Zahl mit gegebener Bitgröße generieren

2. Modulo der Generierten Zahl und der 5000 ersten Primzahlen vergleichen

   (a) Falls Modulo gleich null

   (b) p um zwei erhöhen

   (c) Zurück zu Schritt 2 gehen

3. Fermat-Test durchführen

   (a) Falls fehlgeschlagen

   (b) Zurück zu Schritt 1 gehen

4. Miller-Rabin-Test durchführen

   (a) Falls fehlgeschlagen

   (b) Zurück zu Schritt 1 gehen

5. Primzahl speichern

[2] Normalerweise 1-20 Sekunden

```
// BigInteger ist ein Datentyp welcher beliebig große Ganzzahlen darstellen kann
BigInteger p = 0;

// p und q werden gleichzeitig berechnet, p als t1 und q als t2 (hier ausgelassen)
var t1 = new Task(() =>
{
  // Dies ist ein Label zu welchem jederzeit zurückgegangen werden kann
  Restart:
  // Diese Funktion setzt p auf eine kryptographisch sichere 2048-bit/2056-byte BigInteger-Zahl.
  p = GetRandomBigInt(256);

  // Setze das LSB_0-bit auf 1, da alle Primzahlen größer 2 ungerade sind
  p |= 1 << 0;

  Failed:
  bool failed = false;

  // Parallelisierte Probedivision der zu untersuchenden Zahl durch die ersten 5000 Primzahlen
  Parallel.ForEach(Primes.List, (prime, state) =>
  {
    // Falls p mod prime gleich Null ist ist p eine zusammengesetzte Zahl
    if (p % prime == 0)
    {
      // Wenn p zusammengesetzt ist um zwei erhöhen, die failed-Flagge setzen und aus der parallen Berechnung ausbrechen
      p += 2;
      failed = true;
      state.Break();
    }
  });
  // Falls Probedivision fehlschlug zurück zu Failed gehen und mit dem um zwei erhöhten p erneut versuchen
  if (failed)
    goto Failed;

  // Eine Iteration des Fermat-Testes ausführen und falls dieser fehlschlägt mit einem komplett neuen p beginnen
  if (!Fermat(p, 1))
    goto Restart;
  // 5 Iterationen des Miller-Rabin-Testes ausführen und falls dieser p als Primzahl einstuft die Ausführung der Task beenden
  if (MillerRabin.IsPrime(p, 5))
    return;
  // Falls Miller-Rabin fehlschlägt wieder komplett neu starten
  goto Restart;
```

Abbildung 3.1: Primzahlauswahlprozess in C#

3.3 Verschlüsselung

3.4 Entschlüsselung

3.5 Signatur

4. Auswertung

4.1 Faktorisierung von „RSA-155“ und „RSA-160“

Als Teil meiner Arbeit und als Demonstration der Notwendigkeit von immer größeren Bitgrößen für n habe ich unter Verwendung der „CADO-NFS“ Software, welche eine Implementation des „**N**umber-**F**ield-**Sieve**“-Algorithmus (Zahlkörpersiebalgorithmus) bereitstellt zwei Primfaktorzerlegungen vorgenommen. Für Zahlen größer als 10^{100} ist der Zahlkörpersieb der effizienteste bekannte Algorithmus zur Primfaktorzerlegung[Har]. „RSA-155“ ist eine Fastprimzahl mit 512-bit und somit größer als 10^{100}, welche 1999 von einem Team unter der Leitung von Herman te Riele in acht Monaten faktorisiert wurde[Cav+00]. „RSA-160“ ist eine Fastprimzahl mit 530-bit, welche 2003 von einem Team der Universität Bonn in Zusammenarbeit mit dem Bundesamt für Sicherheit in der Informationstechnik (BSI) faktorisiert wurde. Fünfundzwanzig Jahre später war ich in der Lage „RSA-155“ auf einem Intel Pentium G4400T aus dem Jahre 2015 in nur rund zwanzig Tagen[1] zu faktorisieren. Dies zeigt klar, dass die Faktorisierung von Fastprimzahlen im Laufe der Zeit durch Fortschritte im Bereich der Primfaktorzerlegung und einen breiteren öffentlichen Zugang zu Leistungsstarken Computern und entsprechenden Softwarepaketen um einiges erleichtert wurde. Dadurch werden größere Werte n zwingend nötig, da RSA-Schlüssel sonst in für einen Angreifer realisierbarer Zeit gebrochen werden könnten. Angreifer mit ausreichenden Rechenressourcen, wie zum Beispiel staatlich unterstützte Akteure oder solche mit Cloud- oder Clustercomputern können RSA-Schlüssel mit solch geringer Bitgröße in einer noch kürzeren Zeit brechen. Meine Faktorisierung hat die erwarteten Primfaktoren 1066034883801684548209272203600128786792079585759892915222706082371930628086 43 und 1026395928297411057720541965739916759007165678080380668033419335217907113077 79 geliefert, welche auch 1999 so errechnet wurden:

$$
\begin{aligned}
p &= 1066034883801684548209272203600128786792079585759892915222706082371930628086 43 \\
q &= 1026395928297411057720541965739916759007165678080380668033419335217907113077 79 \\
n = pq &= 1094173864157052742180970732204035761200373294544920599091384213147634998 \\
&\hookrightarrow\ 4288934784717997257891267332497625752899781833797076537244027146743531593354 33 \\
&\hookrightarrow\ 3897
\end{aligned}
\tag{4.1}
$$

[1] 1.77025×10^6 Sekunden entsprechen 20 Tagen 11 Stunden 44 Minuten und 08 Sekunden

4.2 Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel „Bedeutung von starken Primzahlen – Eine Einführung in das RSA-Verschlüsselungssystem“ selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

_____

Ort, Datum Unterschrift

Literatur

[Aic22] Daniela Aichner. „Primzahlen und ihre Bedeutung in der Kryptographie“. Diplomarbeit. Innsbruck, Österreich: Universität Innsbruck, Feb. 2022, S. 45, 48. URL: https://www.uibk.ac.at/mathematik/algebra/media/teaching/diplomarbeit.pdf (besucht am 28.06.2024).

[Bun24] Bundesamt für Sicherheit in der Informationstechnik. *Kryptographische Verfahren: Empfehlungen und Schlüssellängen*. Technische Richtlinie 02102-1. Bundesamt für Sicherheit in der Informationstechnik, Feb. 2024, S. 35. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf (besucht am 09.07.2024).

[Cav+00] Stefania Cavallar u.a. „Factorization of a 512-bit RSA Modulus“. In: *Advances in Cryptology — EUROCRYPT 2000*. Springer Berlin Heidelberg, 2000, S. 1–18. ISBN: 9783540455394. DOI: 10.1007/3-540-45539-6_1. URL: http://dx.doi.org/10.1007/3-540-45539-6_1 (besucht am 14.11.2024).

[con24a] Wikipedia contributors. *Fermat primality test — Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Fermat_primality_test%5C&oldid=1227031378 (besucht am 28.06.2024).

[con24b] Wikipedia contributors. *Miller–Rabin primality test — Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Miller%E2%80%93Rabin_primality_test%5C&oldid=1222212331 (besucht am 28.06.2024).

[con24c] Wikipedia contributors. *RSA (cryptosystem) — Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/w/index.php?title=RSA_(cryptosystem)%5C&oldid=1221958777 (besucht am 28.06.2024).

[Eul] Leonhard Euler. *Theoremata arithmetica nova methodo demonstrata*. URL: https://scholarlycommons.pacific.edu/cgi/viewcontent.cgi?article=1270&context=euler-works (besucht am 18.11.2024).

[Har] Sam Harwell. *What is the fastest integer factorization algorithm?* [Archivierter Link]. URL: https://stackoverflow.com/a/2274520 (besucht am 29.07.2024).

[Kle+10] Thorsten Kleinjung u.a. „Factorization of a 768-Bit RSA Modulus“. In: *Advances in Cryptology – CRYPTO 2010*. Springer Berlin Heidelberg, 2010, S. 333–350. ISBN: 9783642146237. DOI: 10.1007/978-3-642-14623-7_18. URL: http://dx.doi.org/10.1007/978-3-642-14623-7_18 (besucht am 14.11.2024).

[Rab80] Michael Oser Rabin. „Probalistic Algorithm for Testing Primality“. In: *Journal of Number Theory* 12.1 (Feb. 1980), S. 128–138. ISSN: 0022-314X. DOI: 10.1016/0022-314x(80)90084-0. URL: http://dx.doi.org/10.1016/0022-314x(80)90084-0 (besucht am 14.11.2024).

[Rob99] Eric Roberts. *Prime Generation & Testing for primality*. 1999. URL: https://www-cs-faculty.stanford.edu/people/eroberts/courses/soco/projects/1998-99/randomized-algorithms/applications/primality.html (besucht am 28.06.2024).

[RSA78] R. L. Rivest, A. Shamir und L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. 1978. DOI: 10.21236/ada606588. URL: http://dx.doi.org/10.21236/ada606588 (besucht am 14.11.2024).

[Sha49] C. E. Shannon. „Communication theory of secrecy systems“. In: *The Bell System Technical Journal* 28.4 (1949), S. 656–715. DOI: 10.1002/j.1538-7305.1949.tb00928.x. URL: https://ieeexplore.ieee.org/document/6769090 (besucht am 28.06.2024).

[Weia] Eric W. Weisstein. *Carmichael Function*. URL: https://mathworld.wolfram.com/CarmichaelFunction.html (besucht am 20.11.2024).

[Weib] Eric W. Weisstein. *Euler's Totient Theorem*. URL: https://mathworld.wolfram.com/EulersTotientTheorem.html (besucht am 19.11.2024).

[Weic] Eric W. Weisstein. *Fermat's Little Theorem*. URL: https://mathworld.wolfram.com/FermatsLittleTheorem.html (besucht am 10.11.2024).

[Weid] Eric W. Weisstein. *Primary*. URL: https://mathworld.wolfram.com/Primary.html (besucht am 20.11.2024).

[Weie] Eric W. Weisstein. *Totient Function*. URL: https://mathworld.wolfram.com/TotientFunction.html (besucht am 18.11.2024).

[Wol] Wolfram|Alpha. *fermat's little theorem*. URL: https://www.wolframalpha.com/input?i=fermat%27s+little+theorem (besucht am 10.11.2024).

[Zim20] Paul Zimmermann. *Factorization of RSA-250*. Feb. 2020. URL: https://sympa.inria.fr/sympa/arc/cado-nfs/2020-02/msg00001.html (besucht am 14.11.2024).