Universität Siegen

Naturwissenschaftlich-Technische Fakultät

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science Informatik

Evaluierung und Implementierung einer didaktischen Demonstration des RSA-Kryptosystems in CrypTool-Online unter Nutzung moderner Web-Technologien

Betreuer	Dr. Doris Behrendt, Jendrik Winkel
Erstprüfer	Prof. Bernhard Esslinger
Zweitprüfer	Prof. Dr. Roland Wismüller
vorgelegt von	Kevin Stober
Matrikelnummer	1127126
Abgabedatum	30. September 2025 (Update 30.12.2025)

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Evaluierung und Neuimplementierung einer Web-Anwendung zur didaktischen Demonstration des RSA-Kryptosystems im Rahmen des Open-Source-Projekts CrypTool-Online. Ausgangspunkt ist die bestehende Anwendung „RSA visuell und mehr“, welche für eine frühere Version von CrypTool-Online entwickelt wurde und als Grundlage für eine Analyse des didaktischen und technischen Optimierungspotenzials dient. Im Rahmen dieser Arbeit werden zunächst die theoretischen Grundlagen des RSA-Kryptosystems sowie relevanter Web-Technologien ergründet. Darauf aufbauend erfolgt eine detaillierte Evaluierung der bestehenden Implementierung. Anschließend wird eine neue Anwendung unter Einsatz moderner Web-Technologien, wie React und Chakra UI, konzeptioniert und umgesetzt. Diese ermöglicht Lernenden eine interaktive und schrittweise Einarbeitung in zentrale Konzepte des RSA-Kryptosystems. Die Arbeit leistet damit einen Beitrag zur Weiterentwicklung von CrypTool-Online als Lehr- und Lernplattform für Kryptografie.

Abstract

This thesis addresses the evaluation and reimplementation of a web application designed for the didactic demonstration of the RSA cryptosystem within the open-source project CrypTool-Online. The work builds on the existing application “RSA visual and more”, which was developed for an earlier version of CrypTool-Online and serves as the basis for an analysis of its didactic and technical potential for improvement. The thesis first examines the theoretical foundations of the RSA cryptosystem as well as relevant web technologies. On this basis, a detailed evaluation of the existing implementation is carried out. Subsequently, a new application is designed and implemented using modern web technologies such as React and Chakra UI. The resulting application enables learners to engage with core concepts of the RSA cryptosystem in an interactive and step-by-step manner. Overall, this work contributes to the further development of CrypTool-Online as a teaching and learning platform for cryptography.

Inhaltsverzeichnis

1. Einleitung

Asymmetrische Kryptosysteme sind aus dem digitalen Alltag nicht mehr wegzudenken. Von der verlässlichen Authentifizierung im Netz [41] bis hin zur verschlüsselten Kommunikation via E-Mail [52] bilden sie einen unerlässlichen Grundpfeiler der modernen Informationssicherheit. Eines der ältesten und etabliertesten asymmetrischen Kryptosysteme ist das 1978 von Rivest, Shamir und Adleman veröffentlichte RSA-Kryptosystem [42]. Es dient dank seiner zugänglichen mathematischen Grundlagen als beliebtes Beispiel bei der didaktischen Vermittlung von Konzepten rund um die asymmetrische Kryptografie.

1.1. Motivation

Die zahlentheoretischen Grundlagen des RSA-Kryptosystems sind bereits Lernenden ab Sekundarstufe II vermittelbar [51]. Interaktive Anwendungen können eine interessante Ergänzung des Unterrichtsmaterials darstellen und dabei helfen, theoretisches Grundlagenwissen anhand von praktischen Anwendungsszenarien zu vertiefen.

CrypTool-Online (CTO) ist eine browserbasierte Plattform zum Erlernen kryptografischer Konzepte [16], die als Teil des Open-Source-Projekts CrypTool (CT) angeboten wird. Der besondere Vorteil von CTO besteht darin, dass keine zusätzliche Software installiert werden muss. Zahlreiche interaktive Demonstrationen und Visualisierungen zu den verschiedensten kryptografischen Verfahren laden direkt im Browser zum Experimentieren ein. Für das RSA-Kryptosystem existieren in der aktuellen Version von CTO bereits zwei Anwendungen:

1. „RSA (Schritt für Schritt erklärt)“ führt Lernende am Beispiel kleiner Zahlen an das Verfahren heran [18]

2. „OpenSSL“ ermöglicht eine Simulation des Verfahrens im Kontext realistischer Anwendungsszenarien [17]

Darüber hinaus gibt es die Anwendung „RSA visuell und mehr“ [19], welche auf Grundlage einer früheren Version von CTO entstanden ist. Diese setzt sich aus mehreren Unteranwendungen zusammen. Die Unteranwendung „RSA didaktisch“ schlägt eine Brücke zwischen den oben aufgelisteten Anwendungen und demonstriert das Verfahren anhand eines semirealistischen Anwendungsszenarios. Die Vorgänge der Schlüsselgenerierung, des Schlüsselaustauschs sowie der Ver- und Entschlüsselung entsprechen im Kern einem realistischen Anwendungsszenario, werden anders als bei realen Implementierungen des RSA-Kryptosystems jedoch möglichst transparent und nachvollziehbar dargestellt.

1.2. Ziele

Das Hauptziel dieser Arbeit ist die Entwicklung einer Web-Anwendung für CTO, die es Lernenden ermöglicht, eigene RSA-Schlüssel zu generieren, diese untereinander auszutauschen und anschließend kurze Textnachrichten mithilfe des RSA-Verfahrens zu ver- und entschlüsseln. Den Ausgangspunkt für die Entwicklung bildet die bestehende Anwendung „RSA didaktisch“. Diese soll auf ihr technisches und didaktisches Optimierungspotenzial untersucht und unter Verwendung der modernen JavaScript-Bibliothek React in die aktuelle Version von CTO überführt werden. Auch etwaige Fehler im Quellcode der bestehenden Anwendung sollen dabei identifiziert und korrigiert werden.

Die Ausführung der Algorithmen sowie der Aufbau der Benutzeroberfläche soll ausschließlich clientseitig erfolgen. Dabei soll ein besonderes Augenmerk darauf liegen, die Verwendbarkeit der Anwendung auf mobilen Endgeräten sicherzustellen. Bei der Umsetzung der Algorithmen stehen Aspekte der didaktischen Nachvollziehbarkeit statt kryptografischer Robustheit im Vordergrund. Der Fokus liegt auf der Demonstration des reinen RSA-Verfahrens (sogenanntes „Textbook“-RSA) ohne Berücksichtigung zusätzlicher Padding- oder Optimierungsverfahren.

1.3. Aufbau der Arbeit

Der weitere Verlauf der Arbeit umfasst folgende Kapitel:

- Kapitel 2 behandelt die theoretischen Grundlagen des RSA-Kryptosystems sowie relevante technische Grundlagen
- Kapitel 3 beschreibt den Status quo der bestehenden Anwendung und definiert daraus einen Anforderungskatalog
- Kapitel 4 dokumentiert die Implementierung der neuen Anwendung und beleuchtet ausgewählte Implementierungsdetails
- Kapitel 5 fasst die Ergebnisse der Arbeit zusammen und bietet einen Ausblick auf mögliche weiterführende Arbeiten

2. Grundlagen

In Kapitel 2.1 werden die kryptografischen und mathematischen Grundlagen erläutert, die zum Verständnis des RSA-Kryptosystems erforderlich sind. Kapitel 2.2 behandelt zentrale Konzepte der Programmiersprache JavaScript sowie der Bibliotheken React und Chakra UI, insoweit sie für die Implementierung der Anwendung relevant sind.

2.1. RSA (Rivest-Shamir-Adleman)

Bei der Einordnung von Kryptosystemen unterscheidet man zwischen zwei grundlegenden Paradigmen: der symmetrischen und der asymmetrischen Kryptografie [49, S. 185]. Das RSA-Kryptosystem ist ein Vertreter der asymmetrischen Kryptosysteme.

Charakteristisch für die symmetrische Kryptografie (engl.: *secret-key cryptography*) ist, dass derselbe Schlüssel sowohl für die Ver- als auch die Entschlüsselung verwendet wird. Ein zentrales Problem besteht dabei im sicheren Austausch des geheimen Schlüssels (engl.: *secret key*) zwischen zwei Kommunikationspartnern.

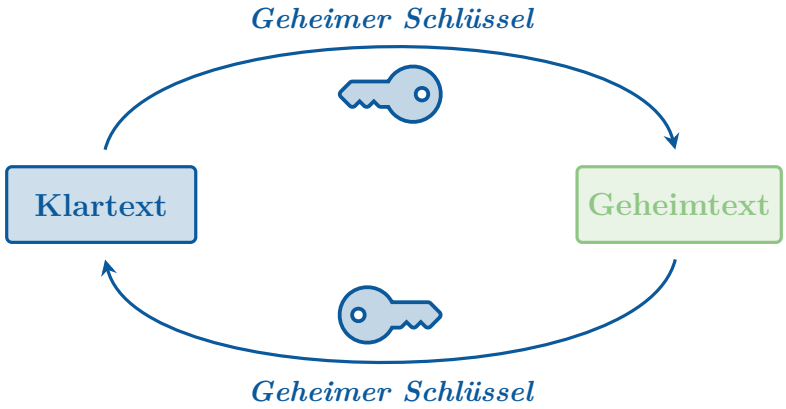

Abbildung 1: Schaubild[1] der symmetrischen Kryptografie in Anlehnung an [47, S. 25]

Die asymmetrische Kryptografie (engl.: *public-key cryptography*) basiert hingegen auf einem Schlüsselpaar, bestehend aus einem öffentlichen Schlüssel (engl.: *public key*) zum Verschlüsseln und einem privaten Schlüssel (engl.: *private key*) zum Entschlüsseln. Da sich der private Schlüssel in der Praxis nicht aus dem öffentlichen Schlüssel berechnen lässt, kann letzterer bedenkenlos veröffentlicht werden. Somit lösen asymmetrische Kryptosysteme das Problem des Schlüsselaustauschs.

[1] Geheime Informationen sind **blau** dargestellt. Öffentliche Informationen sind **grün** dargestellt.

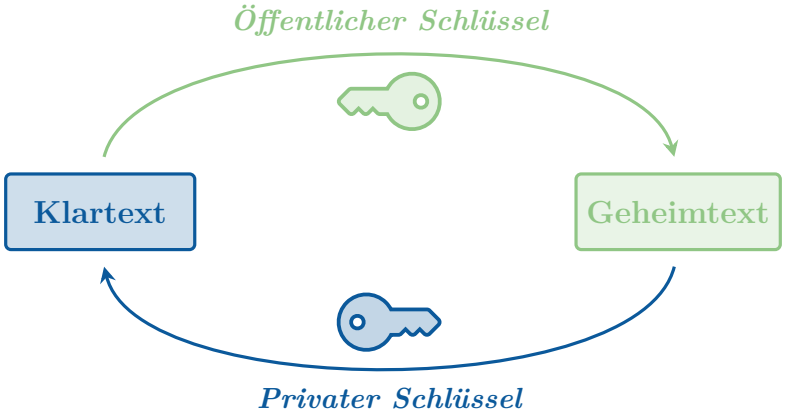

Abbildung 2: Schaubild[2] der asymmetrischen Kryptografie in Anlehnung an [47, S. 25]

Asymmetrische Kryptosysteme sind deutlich rechenintensiver als symmetrische Kryptosysteme [49, S. 186]. In der Praxis kommen aus diesem Grund hybride Verfahren zum Einsatz. Dabei wird zunächst ein zufälliger Sitzungsschlüssel (engl.: *session key*) für ein symmetrisches Kryptosystem generiert. Dieser wird mittels eines asymmetrischen Kryptosystems ausgetauscht. Die Ver- und Entschlüsselung großer Datenmengen erfolgt dann effizient mithilfe des symmetrischen Kryptosystems.

Die Sicherheit von asymmetrischen Kryptosystemen beruht auf dem Konzept von Einwegfunktionen mit Falltür (engl.: *trapdoor one-way functions*) [49, S. 187]. Unter Einwegfunktionen (engl.: *one-way functions*) versteht man Funktionen, die in Polynomialzeit berechenbar, jedoch nicht in Polynomialzeit invertierbar sind. Als Falltür (engl.: *trapdoor*) bezeichnet man eine (geheime) Information, die es ermöglicht, auch die inverse Funktion in Polynomialzeit zu berechnen.

2.1.1. Beschreibung des Verfahrens

Das RSA-Verfahren basiert auf der Annahme, dass eine zusammengesetzte Zahl als Produkt zweier großer Primzahlen in Polynomialzeit berechnet, jedoch nicht in Polynomialzeit wieder in ihre Primfaktoren zerlegt werden kann. Diese Annahme wird auch Faktorisierungsannahme genannt [50, S. 180].

Es sei nachfolgend $\mathbb{N} = \{1, 2, 3, \ldots\}$ definiert als die Menge der natürlichen Zahlen und $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$ definiert als die Menge der natürlichen Zahlen einschließlich 0. Es sei $\mathbb{P}$ definiert als die Menge der Primzahlen.

[2] Geheime Informationen sind **blau** dargestellt. Öffentliche Informationen sind **grün** dargestellt.

Schlüsselgenerierung Die Schlüsselgenerierung des RSA-Kryptosystems [42, S. 122] erfolgt auf Grundlage der Faktorisierungsannahme in folgenden Schritten:

1. Man berechne eine Zahl $n \in \mathbb{N}$ als Produkt zweier großer, zufälliger Zahlen $p, q \in \mathbb{P}$ mit $p \neq q$:
$$n = p \cdot q$$

2. Man wähle eine Zahl $e \in \mathbb{N}$ mit $3 \leq e < n$, sodass:
$$\mathrm{ggT}(e, (p-1) \cdot (q-1)) = 1$$

3. Man bestimme eine Zahl $d \in \mathbb{N}$ mit $1 \leq d < n$ als multiplikatives Inverses von e zum Modul $(p-1) \cdot (q-1)$, das heißt:
$$e \cdot d \equiv 1 \bmod (p-1) \cdot (q-1)$$

Der öffentliche Schlüssel (e, n) besteht aus dem öffentlichen Exponenten e und dem RSA-Modul n. Diese Werte können bedenkenlos veröffentlicht werden.

Der private Schlüssel (d, n) enthält den privaten Exponenten d. Dieser muss geheim gehalten werden. Auch die Werte p, q sowie $(p-1) \cdot (q-1)$ müssen geheim bleiben, da sie es ermöglichen, aus dem öffentlichen Exponenten e den privaten Exponenten d zu berechnen [49, S. 197].

Ver- und Entschlüsselung Die Ver- und Entschlüsselung des RSA-Kryptosystems [42, S. 122] erfolgt auf Grundlage der diskreten Exponentialfunktion mit $x, y, a \in \mathbb{N}_0$ und $b \in \mathbb{N}$:
$$y \equiv x^a \bmod b$$

Es sei $m \in \mathbb{N}_0$ mit $m < n$ ein Klartextrepräsentant und $c \in \mathbb{N}_0$ mit $c < n$ ein Geheimtextrepräsentant (siehe Kapitel 2.1.3 auf Seite 15). Es sei (e, n) ein öffentlicher Schlüssel und (d, n) ein privater Schlüssel gemäß Schlüsselgenerierung.

1. Um einen Klartextrepräsentanten m zu einem Geheimtextrepräsentanten c zu verschlüsseln, verwendet man folgende Verschlüsselungsfunktion unter Einsatz des öffentlichen Schlüssels (e, n):
$$c \equiv m^e \bmod n \tag{1}$$

2. Um einen Geheimtextrepräsentanten c zu einem Klartextrepräsentanten m zu entschlüsseln, verwendet man folgende Entschlüsselungsfunktion unter Einsatz des privaten Schlüssels (d, n):
$$m \equiv c^d \bmod n \tag{2}$$

Die Berechnung der Verschlüsselungsfunktion ist in Polynomialzeit möglich. Es existieren effiziente Verfahren, wie z. B. die binäre Exponentiation [28, S. 132]. Für die Berechnung der inversen Funktion, also die Berechnung von m gegeben c, e und n, besteht die Annahme, dass kein Algorithmus in Polynomialzeit existiert. Diese Annahme wird auch als RSA-Annahme bezeichnet [32, S. 312].

Es liegt auf der Hand, dass die RSA-Annahme höchstens so stark sein kann wie die Faktorisierungsannahme. Sind die Primfaktoren p und q bekannt, kann aus dem öffentlichen Exponenten e gemäß Schlüsselgenerierung das modulare Inverse d berechnet und somit die Verschlüsselungsfunktion invertiert werden. Umgekehrt ist jedoch nicht bewiesen, dass kein Algorithmus zur Invertierung der Verschlüsselungsfunktion existiert, der nicht auf eine Faktorisierung von n hinausläuft. Der Zusammenhang zwischen Faktorisierungs- und RSA-Annahme ist Gegenstand anhaltender Forschung [9, 7, 2, 31].

Unter der RSA-Annahme entspricht die Verschlüsselungsfunktion des RSA-Kryptosystems einer Einwegfunktion. Die Falltür besteht dabei in der Information über die Primfaktorzerlegung von n in Form des privaten Exponenten d. Mit diesem ist auch die Berechnung der inversen Funktion in Polynomialzeit möglich.

Signaturverfahren Das RSA-Kryptosystem eignet sich auch für den Einsatz als Signaturverfahren. Wie in Kapitel 2.1.2 gezeigt, folgt aus den Gleichungen (1) und (2):

$$(m^e)^d \equiv m^{e \cdot d} \equiv m \bmod n$$
$$(m^d)^e \equiv m^{e \cdot d} \equiv m \bmod n$$

Die Ver- und Entschlüsselungsfunktionen sind inverse Abbildungen für alle $m \in \mathbb{N}_0$ mit $m < n$. Es spielt keine Rolle, in welcher Reihenfolge sie auf eine Nachricht m angewandt werden, man erhält immer die ursprüngliche Nachricht m zurück.

Aus dieser Eigenschaft kann folgendes Signaturverfahren konstruiert werden [49, S. 311]:

Es sei $m \in \mathbb{N}_0$ mit $m < n$ eine Nachricht und $s \in \mathbb{N}_0$ mit $s < n$ eine Signatur. Es sei $m_s \in \mathbb{N}_0$ mit $m_s < n$ die aus der Signatur rekonstruierte Nachricht. Es sei (e, n) ein öffentlicher Schlüssel und (d, n) ein privater Schlüssel gemäß Schlüsselgenerierung.

1. Person A berechnet unter Einsatz ihres privaten Schlüssels (d, n) die Signatur s einer Nachricht m:
$$s = m^d \bmod n$$

2. Person B erhält die Signatur s zusammen mit der ursprünglichen Nachricht m.

3. Person B verwendet den öffentlichen Schlüssel (e, n) von Person A, um die Nachricht m_s aus der Signatur s zu rekonstruieren:

$$m_s = s^e \bmod n$$

4. Entspricht m_s der ursprünglichen Nachricht m, so weiß Person B:

   a) Die Nachricht m stammt von Person A.

   b) Die Nachricht m ist unverändert.

2.1.2. Mathematische Grundlagen

Nachfolgend soll gezeigt werden, warum die Ver- und Entschlüsselung des RSA-Kryptosystems funktioniert, also folgende Kongruenz für alle $m \in \mathbb{N}_0$ gilt [24, S. 307–309]:

$$(m^e)^d \equiv (m^d)^e \equiv m^{e \cdot d} \equiv m \bmod n$$

Satz 1 (Satz von Euler-Fermat) *Für alle $a, b \in \mathbb{N}$ mit $\mathrm{ggT}(a, b) = 1$ gilt:*

$$b^{\phi(a)} \equiv 1 \bmod a$$

Der Funktionswert $\phi(a)$ der Eulerschen ϕ-Funktion gibt für jede Zahl $a \in \mathbb{N}$ die Anzahl der positiven natürlichen Zahlen $\leq a$ an, die zu a teilerfremd sind [26, S. 42].

Für alle $p, q \in \mathbb{P}$ gilt folglich:

$$\phi(p) = p - 1 \text{ und } \phi(q) = q - 1$$

Für zusammengesetzte Zahlen $n = p \cdot q$ gilt aufgrund der multiplikativen Eigenschaft von ϕ [26, S. 48]:

$$\phi(n) = \phi(p \cdot q) = \phi(p) \cdot \phi(q) = (p - 1) \cdot (q - 1)$$

Aus Schritt 2 der Schlüsselgenerierung in Kapitel 2.1.1 folgt, dass für e ein multiplikatives Inverses d zum Modul $\phi(n)$ existiert [49, S. 190], das heißt:

$$\begin{aligned} e \cdot d &\equiv 1 \bmod \phi(n) \\ e \cdot d - 1 &\equiv 0 \bmod \phi(n) \end{aligned}$$

Daraus folgt $\phi(n)$ teilt $e \cdot d - 1$ und es existiert ein $k \in \mathbb{N}$, sodass:

$$\begin{aligned} m^{e \cdot d - 1} &= m^{k \cdot \phi(n)} \\ m^{e \cdot d} &= m^{k \cdot \phi(n) + 1} \end{aligned}$$

Nun gilt es, die folgende Kongruenz für alle $m \in \mathbb{N}_0$ zu zeigen:

$$m^{k \cdot \phi(n)+1} \equiv m \bmod n$$

Nach Satz 1 gilt für $\mathrm{ggT}(m, p) = 1$:

$$\begin{aligned} m^{k \cdot \phi(n)+1} &\equiv m^{k \cdot (p-1) \cdot (q-1)+1} \bmod p \\ &\equiv m \cdot \left(m^{p-1}\right)^{k \cdot (q-1)} \bmod p \\ &\equiv m \cdot (1)^{k \cdot (q-1)} \bmod p \\ &\equiv m \bmod p \end{aligned} \tag{3}$$

Sind m und p nicht teilerfremd, also $\mathrm{ggT}(m, p) \neq 1$, dann ist p Teiler von m und es gilt $m^{k \cdot \phi(n)+1} \equiv m \equiv 0 \bmod p$. Die folgende Kongruenz gilt also für alle $m \in \mathbb{N}_0$:

$$m^{k \cdot \phi(n)+1} \equiv m \bmod p \tag{4}$$

Die Gleichungen (4) und (3) gelten analog für q, es gilt also für alle $m \in \mathbb{N}_0$:

$$m^{k \cdot \phi(n)+1} \equiv m \bmod q \tag{5}$$

Aus den Gleichungen (4) und (5) folgt:

$$\begin{aligned} m^{k \cdot \phi(n)+1} - m &\equiv 0 \bmod p \\ m^{k \cdot \phi(n)+1} - m &\equiv 0 \bmod q \end{aligned}$$

Das heißt, p teilt $m^{k \cdot \phi(n)+1} - m$ und q teilt $m^{k \cdot \phi(n)+1} - m$.

Aufgrund von $p, q \in \mathbb{P}$ und $p \neq q$ folgt: $p \cdot q$ teilt $m^{k \cdot \phi(n)+1} - m$ und es gilt für alle $m \in \mathbb{N}_0$ und $n = p \cdot q$:

$$\begin{aligned} m^{k \cdot \phi(n)+1} - m &\equiv 0 \bmod n \\ m^{k \cdot \phi(n)+1} &\equiv m \bmod n \end{aligned}$$

Zusammenfassend gilt für alle $m \in \mathbb{N}_0$:

$$(m^e)^d \equiv (m^d)^e \equiv m^{e \cdot d} \equiv m^{k \cdot \phi(n)+1} \equiv m \bmod n$$

Für alle $m < n$ sind die Ver- und Entschlüsselungsfunktionen bijektive Abbildungen auf $\{0, \ldots, n-1\}$. Für alle $m \geq n$ trifft diese Aussage hingegen nicht zu. Es sei $m = c + n$ mit $c \in \mathbb{N}_0$. Dann gilt:

$$\begin{aligned} m^{e \cdot d} \equiv m \equiv c + n \equiv c \bmod n \\ c^{e \cdot d} \equiv c \bmod n \end{aligned} \tag{6}$$

Es kann also nach einer Ver- und Entschlüsselung nicht mehr zwischen m und c unterschieden werden.

Alternative Schlüsselgenerierung In modernen Standards, wie [37, S. 9] oder [46, S. 35], wird eine Alternative zu der in Kapitel 2.1.1 beschriebenen Schlüsselgenerierung empfohlen. Dabei wird für die Berechnung der Exponenten e und d anstelle des Moduls $\phi(n)$ der Modul $\lambda(n)$ verwendet, wobei λ für die Carmichael-Funktion steht.

Für eine zusammengesetzte Zahl $n = p \cdot q$ berechnet sich diese wie folgt [49, S. 247]:

$$\lambda(n) = \frac{(p-1) \cdot (q-1)}{\text{ggT}(p-1, q-1)} = \text{kgV}(p-1, q-1) \tag{7}$$

Man bestimme also abweichend von Schritt 2 und 3 der Schlüsselgenerierung in Kapitel 2.1.1 den öffentlichen Exponenten e und den privaten Exponenten d, sodass gilt:

$$\text{ggT}(e, \lambda(n)) = 1 \text{ und } e \cdot d \equiv 1 \bmod \lambda(n) \tag{8}$$

Nachfolgend soll gezeigt werden, dass auch diese Schlüsselgenerierung zu einem funktionierenden RSA-Kryptosystem führt, also folgende Kongruenz für alle $m \in \mathbb{N}_0$ gilt:

$$(m^e)^d \equiv (m^d)^e \equiv m^{e \cdot d} \equiv m \bmod n$$

Die Herleitung erfolgt analog zum vorangegangenen Kapitel 2.1.2. Aus Gleichung (8) folgt $\lambda(n)$ teilt $e \cdot d - 1$ und es existiert ein $t \in \mathbb{N}$, sodass:

$$m^{e \cdot d} = m^{t \cdot \lambda(n) + 1}$$

Das heißt, es ist folgende Kongruenz für alle $m \in \mathbb{N}_0$ zu zeigen:

$$m^{t \cdot \lambda(n) + 1} \equiv m \bmod n$$

Es seien:

$$u = \frac{(q-1)}{\text{ggT}(p-1, q-1)} \text{ und } v = \frac{(p-1)}{\text{ggT}(p-1, q-1)}$$

Dann folgt aus Gleichung (7):

$$\lambda(n) = (p-1) \cdot u \text{ und } \lambda(n) = (q-1) \cdot v$$

Analog zu Gleichung (3) gilt nun nach Satz 1 für $\text{ggT}(m, p) = 1$:

$$\begin{aligned} m^{t \cdot \lambda(n)+1} &\equiv m^{t \cdot (p-1) \cdot u+1} \bmod p \\ &\equiv m \cdot (m^{p-1})^{t \cdot u} \bmod p \\ &\equiv m \cdot (1)^{t \cdot u} \bmod p \\ &\equiv m \bmod p \end{aligned} \tag{9}$$

Und für $\text{ggT}(m, q) = 1$:

$$\begin{aligned} m^{t \cdot \lambda(n)+1} &\equiv m^{t \cdot (q-1) \cdot v+1} \bmod q \\ &\equiv m \cdot (m^{q-1})^{t \cdot v} \bmod q \\ &\equiv m \cdot (1)^{t \cdot v} \bmod q \\ &\equiv m \bmod q \end{aligned} \tag{10}$$

Die Herleitung für $\text{ggT}(m, p) \neq 1$ und $\text{ggT}(m, q) \neq 1$ erfolgt analog zum vorangegangenen Kapitel 2.1.2 mit $m^{t \cdot \lambda(n)+1}$ statt $m^{k \cdot \phi(n)+1}$.

Aus den Gleichungen (9) und (10) wird ersichtlich, dass die hinreichende Bedingung zur Anwendung von Satz 1 darin besteht, dass $\lambda(n)$ sowohl ein Vielfaches von $p-1$ als auch ein Vielfaches von $q-1$ ist. Gemäß Gleichung (7) ist $\lambda(n)$ das kleinste solche Vielfache. Bei einer Schlüsselgenerierung unter der Bedingung $e \cdot d \equiv 1 \bmod \lambda(n)$ wird also der kleinstmögliche Exponent d ermittelt.

2.1.3. Sicherheitsaspekte

In der bislang beschriebenen „Reinform“ (auch „Textbook“-RSA genannt [6]) weist das RSA-Kryptosystem eine Reihe von Problemen auf, die einer sicheren Anwendung entgegenstehen. Zwei dieser Probleme sollen hier beispielhaft skizziert werden:

1. Bei Verwendung eines kleinen öffentlichen Exponenten e findet bei der Verschlüsselung $c \equiv m^e \bmod n$ eines Klartextrepräsentanten $m < n^{1/e}$ keine modulare Reduktion statt, das heißt:

   $$m^e \bmod n = m^e$$

   Der Klartextrepräsentant m kann also einfach durch das Ziehen der e-ten Wurzel aus dem Geheimtextrepräsentanten c berechnet werden [25, S. 205].

2. Das RSA-Verfahren ist zudem deterministisch. Das bedeutet, die Verschlüsselung eines Klartextrepräsentanten liefert immer denselben Geheimtextrepräsentanten. Das RSA-Kryptosystem bietet somit keine Sicherheit gegen Chosen-Plaintext-Angriffe. Gelangt ein Angreifer an einen Geheimtextrepräsentanten und ist der vermutete Klartextrepräsentantenraum ausreichend klein, so kann der Angreifer mögliche Klartextrepräsentanten mithilfe des öffentlichen Schlüssels verschlüsseln und diese mit dem gegebenen Geheimtextrepräsentanten vergleichen [49, S. 240].

Für eine sichere Anwendung muss das RSA-Kryptosystem mit sogenannten Padding-Verfahren [37, S. 19, 32] kombiniert werden. Dabei wird die Struktur eines Klartextrepräsentanten vor der Verschlüsselung durch das Hinzufügen zufälliger Bits randomisiert. So kann zum einen sichergestellt werden, dass der zu verschlüsselnde Klartextrepräsentant stets eine ausreichende Länge aufweist, sodass $m \geq n^{1/e}$. Zum anderen wird verhindert, dass die Verschlüsselung eines Klartextrepräsentanten immer denselben Geheimtextrepräsentanten liefert, wodurch das Verfahren nicht länger deterministisch ist.

Klartextrepräsentanten Sollen mit dem RSA-Kryptosystem nicht nur ganze Zahlen sondern beliebige Zeichenfolgen verschlüsselt werden, müssen letztere zunächst als ganze Zahlen codiert werden. Zur Verdeutlichung wird in [37, S. 6–7] zwischen Klartext und Klartextrepräsentanten unterschieden. Als Klartext bezeichnet man die zu verschlüsselnde Zeichenfolge. Als Klartextrepräsentanten bezeichnet man die als natürliche Zahlen codierte Zeichenfolge.

Um einen Klartext als Klartextrepräsentanten zu codieren, eignet sich die b-adische Darstellung [10, S. 170]:

Es sei A ein Alphabet der Länge $l \in \mathbb{N}$ mit $1 < l \leq n$ und n ein RSA-Modul gemäß Kapitel 2.1.1. Es seien die natürlichen Zahlen $0, \ldots, l-1$ die Alphabetzeichen des Alphabets A. Es sei B ein Alphabet derselben Länge l mit beliebigen Alphabetzeichen. Dann können beliebige Alphabetzeichen durch eine bijektive Abbildung $f : B \to A$ als natürliche Zahlen ausgedrückt werden.

Alphabet B	A	B	C	$\cdots$	T	U	V	W	X	Y	Z
Alphabet A	00	01	02	$\cdots$	19	20	21	22	23	24	25

Tabelle 1: Beispiel einer Zuordnung von Alphabetzeichen zu natürlichen Zahlen

Es sei $b \in \mathbb{N}$ mit $b \geq l$ die Basis der b-adischen Darstellung. Es sei $k \in \mathbb{N}$ die Anzahl der Stellen einer Zahl in der b-adischen Darstellung. Dann hat jeder Klartext $m_A = m_1 \ldots m_k$ mit $m_i \in A$ und $1 \leq i \leq k$ einen eindeutigen Klartextrepräsentanten $m \in \mathbb{N}_0$ der Form:

$$m = \sum_{i=1}^{k} m_i \cdot b^{k-i} \tag{11}$$

Um sicherzustellen, dass für jeden Klartextrepräsentanten $m < n$ gilt, wird die maximale Anzahl der Stellen $k_{max} \in \mathbb{N}$ auf Grundlage des RSA-Moduls n sowie der Basis b bestimmt:

$$k_{max} = \lfloor \log_b(n) \rfloor \tag{12}$$

Damit ein Klartextrepräsentant m als Konkatenation der Zahlen m_i dargestellt werden kann, wird als Basis b häufig eine Zehnerpotenz verwendet. Konkret wird die kleinstmögliche Zehnerpotenz verwendet, sodass $b \geq l$.

Es sei im folgenden Beispiel das Alphabet B definiert als das Alphabet der lateinischen Großbuchstaben A–Z mit $l = 26$. Die kleinstmögliche Zehnerpotenz, sodass $b \geq l$ gilt, ist somit $b = 100$. Es sei das Alphabet A definiert als das Alphabet mit den natürlichen Zahlen $0, \ldots, l-1$ als Alphabetzeichen. Der Klartext „CT“ mit $m_1 = 02$ und $m_2 = 19$ (siehe Tabelle 1) lässt sich dann folgendermaßen als Klartextrepräsentant codieren:

$$m = m_1 \cdot b^1 + m_2 \cdot b^0 = 02 \cdot 100 + 19 \cdot 1 = 0219$$

Dies entspricht einer Konkatenation der m_i mit entsprechend vorangestellten Nullen, sodass jedes m_i exakt $\log_{10}(b)$ Stellen hat.

Liegen Alphabetlänge l und Basis b weit auseinander, kann ein beträchtlicher Teil des Klartextrepräsentantenraums der natürlichen Zahlen $0, \ldots, n-1$ niemals als Klartextrepräsentant vorkommen.

Im obigen Beispiel können bei gleichbleibendem m_1 und beliebigem $m_2 \in A$ die Zahlen $0226, \ldots, 0299$ niemals als Klartextrepräsentanten vorkommen. Dies gilt entsprechend für jede Stelle m_i einer solchen Darstellung.

Es ist nicht auszuschließen, dass aufgrund der daraus resultierenden mathematischen Struktur aus einem Geheimtextrepräsentanten partielle Informationen über den Klartextrepräsentanten hergeleitet werden können [49, S. 237].

Um eine solche Struktur zu vermeiden, sollte die Basis b so klein wie möglich gewählt werden. Das heißt, es sollte $b = l$ gelten.

Schlüssellänge Maßgeblich für die Sicherheit des RSA-Kryptosystems ist auch eine adäquate Wahl der Schlüssellänge, also der Länge des Moduls n in bit. Das Bundesamt für Sicherheit in der Informationstechnik (BSI) empfiehlt nach aktuellem Stand eine Schlüssellänge von mindestens 3000 bit [46, S. 35]. Auch das National Institute of Standards and Technology (NIST) empfiehlt eine Schlüssellänge von mindestens 3072 bit für eine Verwendung über das Jahr 2030 hinaus [4, S. 19].

Diese Empfehlungen werden regelmäßig aktualisiert und sind darauf ausgelegt, auch bei fortschreitender Rechenleistung genügend Sicherheit für eine mehrjährige Verwendung

eines Schlüsselpaares zu ermöglichen. Einen Überblick über die bislang größten faktorisierten RSA-Moduln bietet Tabelle 2 [24, S. 307–309].

RSA-Modul	Länge (in bit)	Jahr
RSA-250 [8]	829	2020
RSA-240 [8]	795	2019
RSA-768 [33]	768	2009
RSA-200 [3]	663	2005

Tabelle 2: RSA-Moduln nach Jahr ihrer Faktorisierung

Um sicherzustellen, dass bei der Generierung zweier Primfaktoren p und q von möglichst ähnlicher Bitlänge ein RSA-Modul $n = p \cdot q$ mit einer Bitlänge von exakt $s \in \mathbb{N}$ resultiert, empfiehlt das BSI das Intervall $I := [a, b] \cap \mathbb{N}$ von möglichen Werten für die Primfaktoren p und q wie folgt zu wählen [46, S. 86]:

$$I = [\lceil \frac{2^{s/2}}{\sqrt{2}} \rceil, \lfloor 2^{s/2} \rfloor] \cap \mathbb{N} \tag{13}$$

In gängigen Implementierungen von RSA findet man eine alternative Vorgehensweise. Beim Generieren zweier Primfaktoren p und q der Länge $s/2$ werden jeweils die beiden höchstwertigen Bits auf 1 gesetzt. Somit ist bei einer Multiplikation der beiden Primfaktoren garantiert, dass der resultierende Modul $n = p \cdot q$ exakt die Länge s aufweist. Codebeispiel 1 zeigt eine solche Implementierung in der Kryptografiebibliothek *Libgcrypt* des Projekts *GnuPG*.

```
780 /* Set high order bit to 1, set low order bit to 1.  If we are
781     generating a secret prime we are most probably doing that
782     for RSA, to make sure that the modulus does have the
783     requested key size we set the 2 high order bits. */
784 mpi_set_highbit (prime, nbits-1);
785 if (secret)
786     mpi_set_bit (prime, nbits-2);
787 mpi_set_bit(prime, 0);
```

Codebeispiel 1: libgcrypt/cipher/primegen.c, Zeile 780–787 [40]

Diese Vorgehensweise entspricht einer Approximation des Intervalls I in Gleichung (13). Um die Untergrenze I_{min} eines derart approximierten Intervalls aus einer gegebenen Schlüssellänge s zu konstruieren, muss die Binärzahl $(11)_2 = (3)_{10}$ lediglich $(s/2 - 2)$-mal bitweise nach links verschoben werden.

Sei beispielsweise $s = 16$, dann erhält man für $I_{min} = (11000000)_2$.

Eine bitweise Verschiebung nach links entspricht im Binärsystem einer Multiplikation mit dem Faktor 2. Anders ausgedrückt kann die Approximation der Untergrenze I_{min} also wie folgt formuliert werden:

$$I_{min} = 2^{(s/2)-2} \cdot 3 \tag{14}$$

Dass I_{min} eine geeignete Approximation ist, zeigt die folgende Vereinfachung:

$$\frac{2^{s/2}}{\sqrt{2}} = 2^{(s/2)-(1/2)}$$
$$2^{(s/2)-2} \cdot 3 = 2^{(s/2)-(1/2)} \cdot 2^{(-3/2)} \cdot 3 \approx 2^{(s/2)-(1/2)} \cdot 1{,}0607$$

Die Approximation I_{min} liegt stets über der in Gleichung (13) angegebenen Untergrenze des Intervalls I und garantiert, dass die Bitlänge des Moduls n immer exakt s beträgt. Das approximierte Intervall ist jedoch marginal kleiner als das empfohlene Intervall.

Für solche Fälle werden in [46, S. 86] folgende Kriterien angegeben, um die Konformität eines abweichenden Intervalls mit den technischen Richtlinien des BSI zu bewerten:

1. Für p und q muss das gleiche Intervall $I := [a, b] \cap \mathbb{N}$ genutzt werden.

2. Für das Intervall I muss $|I| \geq 2^{-8}b$ gelten.[3]

Die Approximation erfüllt letzteres Kriterium für alle gängigen Schlüssellängen.

Quantensicherheit Wie in Kapitel 2.1 beschrieben, beruht die Sicherheit von asymmetrischen Kryptosystemen auf dem Konzept von Einwegfunktionen.

Im Falle von RSA basiert diese auf dem Problem der Primfaktorzerlegung. Das derzeit schnellste konventionelle Verfahren der Primfaktorzerlegung ist das allgemeine Zahlkörpersieb (engl.: *general number field sieve*). Die Zeit- und Speicherkomplexität für die Faktorisierung eines RSA-Moduls n wird dabei folgendermaßen abgeschätzt [8, S. 65]:

$$O(e^{(64/9)^{1/3}(\log n)^{1/3}(\log\log n)^{2/3}(1+o(1))})$$

Mit dem Aufkommen von Quantencomputern ist damit zu rechnen, dass gängige Einwegfunktionen in Polynomialzeit invertierbar sein werden. Für das Problem der Primfaktorzerlegung existiert beispielsweise der Shor-Algorithmus, dessen Zeitkomplexität für die Faktorisierung eines RSA-Moduls n wie folgt angegeben wird [45, S. 317]:

$$O((\log n)^2(\log\log n)(\log\log\log n))$$

[3] $|I|$ steht dabei für die Anzahl der Elemente in I.

Nach derzeitigem Stand gibt es keine kryptografisch relevante Implementierung des Shor-Algorithmus [48]. In Zukunft könnte das RSA-Kryptosystem jedoch von Quantencomputern gebrochen werden. Ein Fokus der jüngsten Forschung liegt daher neben völlig neuen quantensicheren Verfahren [47, S. 25–35] auch auf quantengehärteten Abwandlungen des RSA-Kryptosystems [5] zur übergangsweisen Verwendung.

2.2. Web-Technologien

Wie bereits die bestehende Anwendung wird auch die im Rahmen dieser Arbeit entstehende Anwendung unter Einsatz der Programmiersprache JavaScript (JS) realisiert. Diese wird im Standard ECMA-262 [30] unter dem Namen ECMAScript spezifiziert. Anders als die bestehende Anwendung basiert die neue Anwendung jedoch auf der modernen JS-Bibliothek React [36], die durch die Komponentenbibliothek Chakra UI [1] ergänzt wird. Im Folgenden werden zentrale Konzepte dieser Technologien vorgestellt.

2.2.1. JavaScript

JS ist eine dynamisch typisierte Programmiersprache. Im Gegensatz zu statisch typisierten Sprachen wird der Datentyp einer Variablen nicht zur Kompilierzeit festgelegt. Stattdessen arbeitet JS mit typisierten Werten. Der Datentyp einer Variablen ergibt sich aus dem ihr zugewiesenen Wert und kann sich während der Laufzeit beliebig ändern.

Datentypen In JS wird zwischen primitiven und nichtprimitiven Datentypen unterschieden [30, S. 6]. Zu den primitiven Datentypen zählen [30, S. 25–42]:

- `undefined`: repräsentiert Variablen ohne zugewiesenen Wert
- `null`: repräsentiert Variablen mit explizit leerem Wert
- `boolean`: Wahrheitswerte der booleschen Aussagenlogik (`true` und `false`)
- `string`: Textdaten als Folgen von UTF-16-Codeeinheiten
- `symbol`: kollisionsfreie Schlüsselsymbole für Schlüssel-Wert-Paare
- `number`: Gleitkommazahlen mit doppelter Genauigkeit (64-Bit IEEE 754)
- `bigint`: Ganzzahlen mit arbiträrer Genauigkeit[4]

[4] Das heißt: Ganzzahlen mit arbiträrer Bitlänge, also von beliebiger Größe

Primitive Datentypen repräsentieren unveränderliche (engl.: *immutable*) Werte. Diese zeichnen sich dadurch aus, dass ihr Inhalt nicht geändert werden kann, ohne dass ein völlig neuer primitiver Wert erzeugt wird. Vergleiche erfolgen nach Wert, sodass zwei Variablen mit demselben primitiven Wert als gleich gelten (siehe Codebeispiel 2).

```
1 let a = 42
2 let b = 42
3 console.log(a === b) // true
```

Codebeispiel 2: Vergleich primitiver Datentypen in JS

Neben den primitiven Datentypen gibt es in JS den nichtprimitiven Datentyp `object`, der eine Sammlung von Schlüssel-Wert-Paaren repräsentiert [30, S. 42–52]. Als Wert eines Schlüssel-Wert-Paares kann wiederum ein Wert vom Datentyp `object` übergeben werden. So können komplexe Datenstrukturen definiert werden. Zum Datentyp `object` gehören in JS unter anderem auch Funktionen und Arrays.

Nichtprimitive Datentypen repräsentieren veränderliche (engl.: *mutable*) Werte, deren Inhalt nachträglich geändert werden kann. So kann beispielsweise der Inhalt eines Arrays verändert werden, ohne ein völlig neues Array zu erzeugen. Vergleiche erfolgen nach Referenz. Zwei Variablen mit demselben nichtprimitiven Wert sind nicht gleich, solange sie nicht dieselbe Referenz teilen (siehe Codebeispiel 3).

```
1 let a = {x: 42}
2 let b = {x: 42}
3 console.log(a === b) // false
4
5 b = a // b verweist auf Referenz von a
6 console.log(a === b) // true
```

Codebeispiel 3: Vergleich nichtprimitiver Datentypen in JS

Umwandlung von Datentypen JS ist eine schwach typisierte Programmiersprache. Darunter versteht man eine Sprache, die zur Laufzeit implizite Umwandlungen von Werten eines Datentyps zu Werten eines anderen Datentyps zulässt. In JS gibt es also zwei Szenarien, in denen eine Umwandlung von Datentypen erfolgt [30, S. 63–73]:

- Explizite Umwandlung: Dabei wird die Umwandlung von einem Datentyp zum anderen explizit im Programmcode deklariert.
- Implizite Umwandlung: Die Umwandlung erfolgt automatisch bei der Auswertung von bestimmten Operatoren bzw. Ausdrücken.

Einer der häufigsten Fälle einer impliziten Umwandlung tritt auf, wenn eine Variable als Teil eines booleschen Ausdrucks ausgewertet wird.

```
1 let a = 42
2 if (a) ...
```

Codebeispiel 4: Implizite Umwandlung zum Datentyp `boolean` in JS

In Codebeispiel 4 wird in Zeile 2 der Wert vom Datentyp `number` in Variable `a` zur Auswertung der `if`-Bedingung implizit in einen Wert vom Datentyp `boolean` umgewandelt [30, S. 64, 303]. Dies entspricht in diesem Fall dem Wert `true`.

Bei der Umwandlung von Werten beliebiger Datentypen in Werte des Datentyps `boolean` wird in JS zwischen sogenannten *truthy* und *falsy* Werten unterschieden. Als *falsy* bezeichnet man Werte, die in einem booleschen Kontext als `false` interpretiert werden. Dazu zählen ausschließlich die in Tabelle 3 aufgeführten Werte [30, S. 64].

Datentyp	*falsy* Werte
`boolean`	`false`
`number`	`0`, `-0`, `NaN`
`bigint`	`0n`
`string`	`""` (leerer String)
`undefined`	`undefined`
`null`	`null`

Tabelle 3: Datentypen und deren *falsy* Werte in JS

Alle nicht in Tabelle 3 aufgeführten Werte zählen zu den *truthy* Werten – werden also in einem booleschen Kontext als `true` interpretiert. Dies umfasst insbesondere alle Werte der Datentypen `object` und `symbol`.

Ein weiterer Fall der impliziten Umwandlung ergibt sich bei der Auswertung von arithmetischen bzw. vergleichenden Operatoren mit Operanden verschiedener Datentypen.

```
1 let a = 42
2 let b = [43]
3 console.log(a < b) // true
4
5 let b = [43, 44]
6 console.log(a < b) // false
```

Codebeispiel 5: Implizite Umwandlung zum Datentyp `number` in JS

In Codebeispiel 5 soll in Zeile 3 der Wert in Variable `b` vom Datentyp `object` (konkret ist dies hier ein Array) mit dem Wert in Variable `a` vom Datentyp `number` verglichen werden. Dies erfolgt über eine für jedes Objekt definierte Methode zur Umwandlung in einen primitiven Datentyp [30, S. 63–64]. In diesem Fall wird der Wert in Variable `b` zunächst gemäß der für Arrays spezifizierten Methode `toString()` als ein Wert (`"43"`) vom primitiven Datentyp `string` ausgegeben [30, S. 595, 608]. Dieser Wert wird schließlich gemäß der Spezifizierung des Vergleichsoperators `<` in einen Wert (`43`) vom Datentyp `number` umgewandelt, um ihn mit dem Wert (`42`) vom Datentyp `number` in Variable `a` zu vergleichen [30, S. 76–77, 279].

In Zeile 6 wird das Array zunächst wieder in einen Wert (`"43,44"`) vom Datentyp `string` umgewandelt. Die anschließende Umwandlung zum Datentyp `number` ergibt `NaN`. Der Vergleich mit `NaN` ergibt immer `false` [30, S. 77, 279].

Das vorangegangene Beispiel verdeutlicht, welche Komplexität die implizite Umwandlung von Datentypen in JS annehmen kann. Zwar erlaubt dieser Mechanismus eine flexible Verwendung von Variablen, ohne Datentypen explizit umwandeln zu müssen. Jedoch birgt das mitunter komplexe Zusammenspiel zwischen Operatoren und Operanden unterschiedlicher Datentypen stets das Risiko unvorhergesehener Konsequenzen. Beim Programmieren ist daher ein hohes Maß an Voraussicht über die beteiligten Datentypen sowie eine entsprechende Kenntnis der relevanten Spezifikationen vonnöten.

Langzahlarithmetik Der primitive Datentyp `number` wird in JS als Gleitkommazahl im Format `binary64` gemäß IEEE 754-2019 dargestellt [30, S. 31]. Eine solche Gleitkommazahl setzt sich aus den folgenden Bestandteilen zusammen [23, S. 23]:

- 1 Bit für das Vorzeichen
- 11 Bits für den Exponenten
- 52 Bits für die Mantisse

Der Mantisse wird ein implizites Bit vorangestellt, das über den Exponenten codiert wird [23, S. 19]. Ganzzahlen können so ohne Verlust von Genauigkeit in einem Bereich von $-(2^{53}-1)$ bis $2^{53}-1$ dargestellt werden [30, S. 456–457].

Wie in Kapitel 2.1.3 auf Seite 16 erläutert, bewegen sich die empfohlenen Schlüssellängen für das RSA-Kryptosystem weit jenseits der obigen Begrenzung von 53 bit. Für eine möglichst realistische Implementierung des Verfahrens ist also die Verarbeitung von großen Ganzzahlen unerlässlich.

Mit ECMAScript 2020 wurde der primitive Datentyp `bigint` spezifiziert [30, S. vi]. Dieser erlaubt die Darstellung von beliebig großen Ganzzahlen, deren Genauigkeit nur vom

verfügbaren Speicher in der Laufzeitumgebung begrenzt wird. In CTO ist für einige Anwendungen, wie „RSA (Schritt für Schritt erklärt)“, bereits die Bibliothek BigInteger.js von Peter Olson [39] eingebunden. Aus diesem Grund bietet sich deren Verwendung auch für die im Rahmen dieser Arbeit entstehende Anwendung an.

BigInteger.js implementiert große Ganzzahlen als Objekte, die über den Konstruktor `bigInt()` initialisiert werden (nicht zu verwechseln mit Werten des primitiven Datentyps `bigint`, die über den Konstruktor `BigInt()` initialisiert werden).

Zu beachten ist dabei, dass die gemäß ECMAScript spezifizierten Arithmetik- bzw. Vergleichsoperatoren nur für Werte primitiver Datentypen, wie `number` oder `bigint`, definiert sind [30, S. 76–78, 287]. Werte nichtprimitiver Datentypen, wie `bigInt`-Objekte, werden bei der Verwendung solcher Operatoren implizit in einen primitiven Datentyp umgewandelt. Dies geht bei einer Umwandlung von Ganzzahlen kleiner als $-(2^{53}-1)$ bzw. größer als $2^{53}-1$ zum primitiven Datentyp `number` zwangsläufig mit einem Verlust von Informationen einher.

ECMAScript	BigInteger.js
`a + b`	`bigInt(a).add(b)`, `bigInt(a).plus(b)`
`a - b`	`bigInt(a).subtract(b)`, `bigInt(a).minus(b)`
`a * b`	`bigInt(a).multiply(b)`, `bigInt(a).times(b)`
`a / b`	`bigInt(a).divide(b)`, `bigInt(a).over(b)`
`a % b`	`bigInt(a).mod(b)`
`a ** b`	`bigInt(a).pow(b)`
`a == b`	`bigInt(a).equals(b)`, `bigInt(a).eq(b)`
`a > b`	`bigInt(a).greater(b)`, `bigInt(a).gt(b)`
`a >= b`	`bigInt(a).greaterOrEquals(b)`, `bigInt(a).geq(b)`
`a < b`	`bigInt(a).lesser(b)`, `bigInt(a).lt(b)`
`a <= b`	`bigInt(a).lesserOrEquals(b)`, `bigInt(a).leq(b)`

Tabelle 4: Operatoren gemäß ECMAScript-Spezifikation und entsprechende Methoden in BigInteger.js

Um eine solche Umwandlung zu verhindern, müssen konsequent die in BigInteger.js definierten Methoden des `bigInt`-Objekts verwendet werden (siehe Tabelle 4). Für viele der Methoden existieren Aliasse, die entweder kürzer ausfallen oder die Formulierung von Ausdrücken ermöglichen, die sich an der natürlichen Sprache orientieren. Jede der Methoden gibt wiederum ein `bigInt`-Objekt zurück, sodass die Methoden verkettet werden können (siehe Codebeispiel 6).

```
137 /**
138 * Calculates Euler's phi function for two primes p, q.
139 *
140 * @param {bigInt.BigNumber} p - Prime p
141 * @param {bigInt.BigNumber} q - Prime q
142 * @returns {bigInt.BigInteger} phi(pq)
143 */
144 const eulersPhi = (p, q) => {
145   return bigInt(p).minus(1).times(bigInt(q).minus(1))
146 }
```

Codebeispiel 6: rsa-didactic/rsaAlgorithm.js, Zeile 137–146

Die Verwendung der Bibliothek BigInteger.js bietet im Vergleich zur Verwendung des primitiven Datentyps `bigint` einige Vorteile. So sind viele für das RSA-Kryptosystem relevante Methoden, wie `modPow()` zur Berechnung der diskreten Exponentialfunktion oder `modInv()` zur Berechnung des modularen Inversen, bereits implementiert. Auch sind mit `isPrime()` und `isProbablePrime()` effiziente Methoden für Primzahltests auf Grundlage des Miller-Rabin-Algorithmus vorhanden [39].

Bibliothek	Performance (in Operationen/s)
Peter Olson BigInteger.js [39]	$62{,}1 \pm 1{,}04\,\%$ (14 Samples)
GoogleChromeLabs JSBI [27]	$13{,}0 \pm 0{,}77\,\%$ (8 Samples)
Yaffle BigInteger [53]	$6{,}98 \pm 0{,}38\,\%$ (6 Samples)

Tabelle 5: Benchmark[5] zur Berechnung von 12345^{12345} in BigInteger-Bibliotheken [38]

Im Hinblick auf die Performance hebt sich die Bibliothek BigInteger.js von Peter Olson in Benchmarks deutlich von Alternativen, wie JSBI von GoogleChromeLabs, ab (siehe Tabelle 5). Unterstützt die jeweilige Laufzeitumgebung den primitiven Datentyp `bigint`, greift BigInteger.js intern auf diesen zurück. Dadurch profitiert die Bibliothek von allen Optimierungen, die bei der Implementierung von großen Ganzzahlen in der jeweiligen Laufzeitumgebung vorgenommen wurden.

[5] Berechnet in Mozilla Firefox Version 142.0.1 (64-bit) unter MacOS 15.5. Längere Balken entsprechen besserer Performance.

2.2.2. React

React ist eine JS-Bibliothek für die Entwicklung von komponentenbasierten User Interfaces (UIs) [36]. Die Bibliothek verfolgt einen deklarativen Ansatz: Anstatt Änderungen im Document Object Model (DOM) – der Programmierschnittstelle zur Repräsentation und Manipulation von HTML bzw. XML-Dokumenten im Browser [29] – imperativ vorzugeben, werden lediglich die möglichen Zustände einer UI definiert. Die Berechnung und Umsetzung der notwendigen Änderungen im DOM übernimmt React [34, S. 7–9].

Zum Rendern einer UI repräsentiert React diese zunächst in einem virtuellen DOM. Bei jeder Zustandsänderung wird eine neue Version dieser Repräsentation erzeugt und mithilfe eines heuristischen Diff-Algorithmus [11] mit der vorherigen Version verglichen. Auf Basis dieses Vergleichs ermittelt React die minimal notwendigen Änderungen und setzt diese gebündelt im realen DOM um. Dieses Vorgehen ermöglicht insbesondere bei komplexen und dynamischen UIs eine Verbesserung der Performance [34, S. 13–14].

Komponenten In React beschreibt eine Komponente einen in sich abgeschlossenen, wiederverwendbaren Teil einer UI. Dabei kann es sich sowohl um grundlegende Elemente, wie Überschriften oder Eingabefelder, als auch um umfassendere Strukturen, wie Formulare oder Unterreiter, handeln. React-Anwendungen bestehen aus einer Hierarchie solcher Komponenten, wobei komplexere Komponenten aus einfacheren Komponenten zusammengesetzt werden. Dieses Kompositionsprinzip fördert eine klare Strukturierung von UIs und erleichtert die Wiederverwendbarkeit von UI-Bausteinen [34, S. 31–34].

Mithilfe einer optionalen JavaScript Syntax Extension (JSX) können Komponentenstrukturen innerhalb von JS-Dateien in einer an HTML- bzw. XML-Syntax angelehnten Form definiert werden [35]. Während klassische Paradigmen der Web-Entwicklung eine Trennung von Struktur (HTML) und Logik (JS) vorsehen, ermöglicht React mit JSX eine übersichtliche Kapselung beider Aspekte innerhalb einer Datei [34, S. 41–45].

```
1 function ErrorDetails(props) {
2   return (
3     <details>
4       <summary>{props.error.name}</summary>
5       {props.error.message}
6     </details>
7   )
8 }
```

Codebeispiel 7: Definition einer Funktionskomponente in React mit JSX

Der Datenfluss innerhalb einer Komponentenhierarchie erfolgt in React grundsätzlich unidirektional. Elternkomponenten können Daten über ein Props-Objekt an Kindkom-

ponenten weitergeben. Dieses Objekt ist innerhalb der Kindkomponenten unveränderlich. Für eine Änderung der Daten innerhalb von Kindkomponenten muss stets ein neues Props-Objekt durch die Elternkomponenten übergeben werden [34, S. 34–35].

```
function ErrorParent() {
  const errorObject = { name: "...", message: "..." }
  return (
    <ErrorDetails error={errorObject} />
  )
}
```

Codebeispiel 8: Aufruf einer Funktionskomponente in React mit JSX

Codebeispiel 7 zeigt die Definition einer Komponente `ErrorDetails` zur Darstellung von Fehlerinformationen. Codebeispiel 8 demonstriert einen Aufruf der Komponente `ErrorDetails` mittels JSX sowie die Übergabe des `error`-Props anhand einer an HTML- bzw. XML-Attribute angelehnten Syntax.

Hooks Um in Funktionskomponenten auf die Zustands- und Lebenszykluslogik von React zugreifen zu können, stellt die Bibliothek sogenannte Hooks bereit [36]:

- `useState()`: Mit diesem Hook können Zustandsvariablen innerhalb von Komponenten verwaltet werden. Der Hook kann mit einem optionalen Startwert initialisiert werden und gibt eine Zustandsvariable sowie eine Änderungsmethode zurück.

```
const [name, setName] = useState("")
```

Codebeispiel 9: Der React-Hook `useState()`

- `useContext()`: Dieser Hook stellt Zustandsvariablen in einer mehrstufigen Komponentenhierarchie bereit, ohne dass diese über mehrere Stufen von der Ursprungskomponente bis hin zur Zielkomponente als Props weitergegeben werden müssen.

- `useEffect()`: Dieser Hook ermöglicht die Behandlung von Seiteneffekten, etwa für die Synchronisierung mit externen Systemen.

```
useEffect(() => {
  document.title = `Hello, ${name}!`
}, [name])
```

Codebeispiel 10: Der React-Hook `useEffect()`

Die Funktion `useEffect(setup, dependencies)` erhält als erstes Argument eine Funktion, die den Seiteneffekt beschreibt. Das zweite Argument gibt ein Array von

Variablen an, bei deren Veränderung der Seiteneffekt ausgeführt werden soll. Für dieses Argument kommen folgende Werte in Frage:

  - Kein Wert: Der Seiteneffekt wird bei jedem Rendern der Komponente ausgeführt.
  - `[]` (leeres Array): Der Seiteneffekt wird nur beim ersten Rendern der Komponente ausgeführt.
  - `[a, b, c, ...]`: Der Seiteneffekt wird immer dann ausgeführt, wenn sich eine der Variablen `a`, `b`, `c`, `...` verändert hat.

- `useMemo()` und `useCallback()`: Mit diesen Hooks kann die Performance optimiert werden, indem Berechnungsergebnisse bzw. Funktionsreferenzen zwischengespeichert und nur bei relevanten Änderungen neu berechnet bzw. erzeugt werden.

2.2.3. Chakra UI

In CTO wird React durch die Komponentenbibliothek Chakra UI [1] ergänzt. Diese stellt eine Vielzahl vordefinierter UI-Bausteine bereit und unterstützt eine einheitliche Gestaltung der UI über alle Anwendungen in CTO hinweg. Zudem erleichtert Chakra UI die barrierefreie Gestaltung von Anwendungen, um sie möglichst vielen Lernenden unabhängig von individuellen Einschränkungen zugänglich zu machen.

Designsystem Im Kern von Chakra UI steht das Konzept eines Designsystems. Ziel dieses Konzepts ist es, wiederkehrende Gestaltungsentscheidungen zu vereinheitlichen und ihre Einhaltung zu vereinfachen. Ein fundamentales Element sind Style Props [15].

```
1  function SummaryLabel(props) {
2    return (
3      <span
4        style={{
5          fontSize: "0.875rem",
6          lineHeight: 1.5,
7          fontWeight: 500,
8        }}
9        {...props}
10     />
11   )
12 }
```

Codebeispiel 11: Komponente ohne Chakra UI

Codebeispiel 11 zeigt eine Komponente ohne Verwendung von Chakra UI. In diesem Fall müssen konkrete Darstellungswerte, wie Schriftgröße oder Zeilenhöhe, explizit festgelegt werden. Die visuelle Ausprägung der Komponente ist dadurch unmittelbar an spezifische CSS-Eigenschaften und -Werte gebunden.

```
function SummaryLabel(props) {
  return <Span textStyle="sm" fontWeight="medium" {...props} />
}
```

Codebeispiel 12: Komponente mit Chakra UI

Im Gegensatz dazu beschreibt die Variante mit Chakra UI die Gestaltung auf einer höheren Abstraktionsebene. Anstelle der konkreten CSS-Eigenschaften `fontSize` und `lineHeight` wird in Codebeispiel 12 die semantische Style Prop `textStyle` verwendet.

Ein wichtiges Konzept in diesem Zusammenhang sind semantische Bezeichner (engl.: *semantic tokens*). Gestaltungseigenschaften werden hierbei nicht über feste CSS-Werte, sondern über konzeptionelle Bezeichner definiert [14].

```
<Icon color={invalid ? "fg.error" : "fg.success"}>
  {invalid ? <FaXmark /> : <FaCheck />}
</Icon>
```

Codebeispiel 13: Verwendung semantischer Bezeichner in Chakra UI

In Codebeispiel 13 wird die Farbe eines Icons abhängig vom Zustand `invalid` über die semantischen Bezeichner `fg.error` und `fg.success` gesteuert. Diese stehen jeweils für einen gültigen bzw. ungültigen Zustand. Die konkrete Ausgestaltung dieser Zustände erfolgt zentral über das Designsystem.

Barrierefreiheit Neben der Gestaltung erleichtert Chakra UI auch die Verbesserung der Barrierefreiheit. Die vordefinierten Komponenten sind standardmäßig auf eine zugängliche Nutzung ausgelegt. Die zugrundeliegende Bibliothek Ark UI unterstützt unter anderem eine konsistente Tastaturnavigation sowie sinnvolle ARIA-Attribute [44].

Darüber hinaus erleichtert Chakra UI die Umsetzung responsiver Layouts [13]. Über deklarative Style Props kann das Layout an unterschiedliche Bildschirmgrößen angepasst werden, ohne eine explizite Definition von Media Queries [43] zu erfordern.

Codebeispiel 14 zeigt eine Layoutanpassung ohne Chakra UI, die über Media Queries realisiert wird. Für Bildschirme mit einer Breite von bis zu 768 px wird eine vertikale Anordnung (`column`) der Elemente definiert, während auf größeren Bildschirmen eine horizontale Anordnung (`row`) beibehalten wird.

```
/* JSX */

<div className="flex" {...props} />

/* CSS */

.flex {
  display: flex;
  flex-direction: row;
}

@media screen and (max-width: 768px) {
  .flex {
    flex-direction: column;
  }
}
```

Codebeispiel 14: Responsive Layoutanpassung ohne Chakra UI

```
<Flex flexDir={{ mdDown: "column" }} {...props} />
```

Codebeispiel 15: Responsive Layoutanpassung mit Chakra UI

Chakra UI abstrahiert diesen Ansatz durch vordefinierte Breakpoints und entsprechende Style Props, die automatisch in passende Media Queries übersetzt werden (siehe Codebeispiel 15). Dies reduziert den Implementierungsaufwand, verbessert die Lesbarkeit des Quellcodes und unterstützt die in Kapitel 1 formulierte Anforderung einer optimierten Verwendbarkeit auf mobilen Endgeräten.

3. Evaluierung

In Kapitel 3.1 wird zunächst der Status quo der bestehenden Anwendung beschrieben. Darauf aufbauend folgt in Kapitel 3.2 ein Anforderungskatalog für die Implementierung der neuen Anwendung, in dem sowohl funktionale als auch didaktische Gesichtspunkte berücksichtigt werden.

3.1. Status quo

Die bestehende Anwendung ist als Teil der Anwendung „RSA visuell und mehr“ [19] unter dem Tab „RSA didaktisch“ zu finden. Sie unterteilt sich wiederum in die drei Tabs: „Schlüsselgenerierung“, „Verschlüsselung“ und „Schlüssel“.

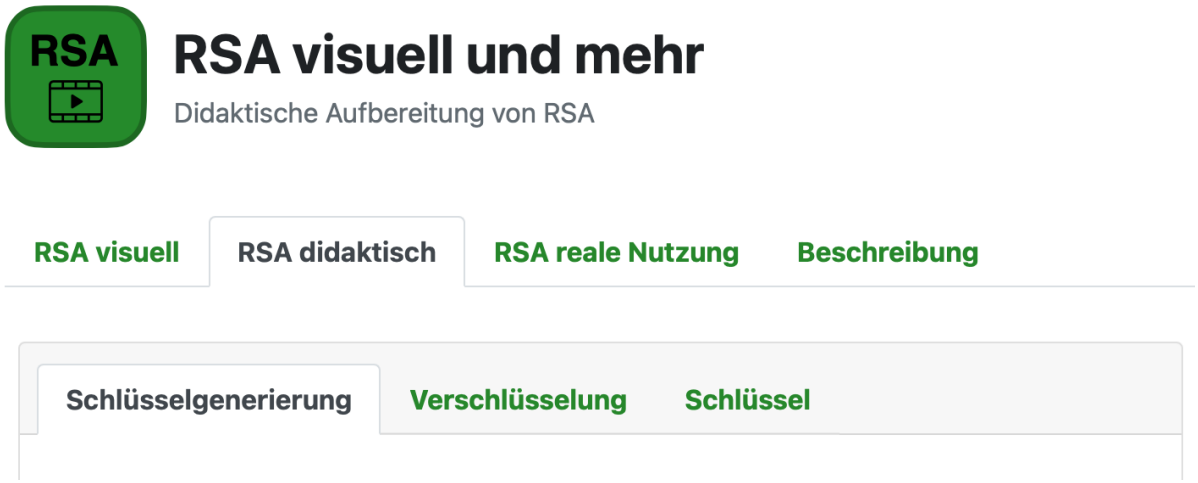

Abbildung 3: Übersicht über die Anwendung „RSA visuell und mehr“ [19]

3.1.1. Tab „Schlüsselgenerierung“

Unter dem Tab „Schlüsselgenerierung“ (für eine Gesamtansicht siehe Abbildung A.1 auf Seite 59) steht an erster Stelle die Schaltfläche „Zufälliges Schlüsselpaar generieren“. Diese öffnet einen Dialog mit den Eingabefeldern „Minimum“ und „Maximum“ sowie der Schaltfläche „Generierung starten“ (siehe Abbildung 4).

Die Eingabe von ungültigen Werten für „Minimum“ bzw. „Maximum“, beispielsweise alphanumerische Eingaben oder Minimum > Maximum, wird mit der generischen

Fehlermeldung „Der Zahlenbereich ist nicht gültig“ quittiert. Bei der Angabe eines Intervalls, das weniger als zwei Primzahlen enthält, erfolgt nach einer Interaktion mit der Schaltfläche „Generierung starten“ eine Zeitüberschreitung im Browser. Die gesamte Seite reagiert nicht mehr auf Eingaben und muss neu geladen werden.

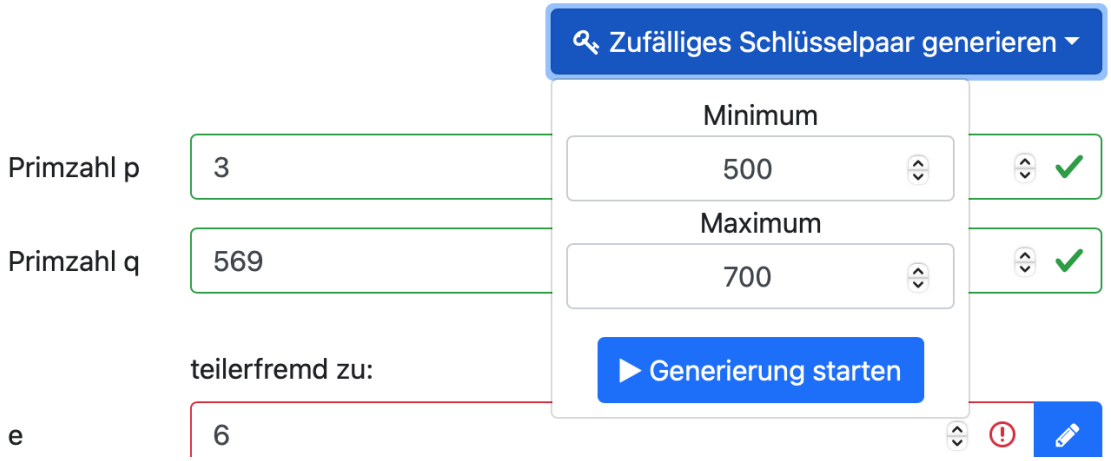

Abbildung 4: Optionen für „Zufälliges Schlüsselpaar generieren“ in [19]

Alternativ kann ein Schlüssel manuell erzeugt werden. Dazu können zwei Primzahlen p und q sowie ein öffentlicher Exponent e eingegeben werden. Letzterer kann zudem über eine Schaltfläche automatisch generiert werden. Der Exponent e kann wahlweise teilerfremd zu $\phi(n)$ oder $\lambda(n)$ (hier unter der Bezeichnung „kgV“) bestimmt werden.

Bei der manuellen Eingabe von Werten für p, q und e wird eine gültige bzw. ungültige Eingabe durch ein Symbol im Eingabefeld angezeigt (grünes Häkchen oder rotes Ausrufezeichen). Außerdem erscheint ein temporäres Feedback mit weiteren Informationen zur Gültigkeit bzw. Ungültigkeit der Eingabe. Auffällig ist hierbei, dass bei einer Eingabe von negativen Zahlen oder Zahlen mit Dezimaltrennzeichen sowie alphanumerischen Eingaben keinerlei Validierung stattfindet (siehe Abbildung 5).

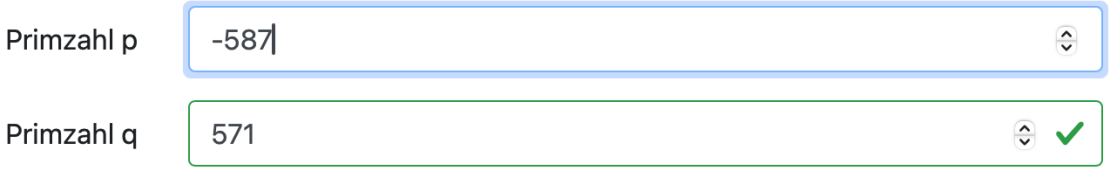

Abbildung 5: Validierung von Eingaben in [19]

Nach den Eingabefeldern für p, q und e folgt ein weiteres Eingabefeld mit der Beschriftung „Name“. Hierüber kann festgelegt werden, unter welchem Namen das generierte Schlüsselpaar gespeichert werden soll.

In einem separaten Abschnitt werden die Werte „kgV“, $\phi(n)$, (d, n) und (e, n) nach erfolgreicher Generierung zusammenfassend aufgeführt.

Über eine Schaltfläche am unteren Ende des Tabs kann ein gültiges Schlüsselpaar für die Verwendung innerhalb der weiteren Tabs gespeichert werden. Eine zweite Schaltfläche bietet die Möglichkeit, den privaten bzw. den öffentlichen Schlüssel separat abzuspeichern. Hierbei fällt auf, dass bei einer Interaktion mit dieser Schaltfläche wider Erwarten nichts geschieht. Nur bei einer Interaktion mit dem kleinen Teil rechts, der durch einen Pfeil gekennzeichnet ist, öffnen sich die weiteren Optionen. Diese sind in der deutschen Übersetzung zudem abgeschnitten (siehe Abbildung 6). Die Benennung des Schlüsselspeichers (engl.: *keystore*) ist in der deutschen Übersetzung nicht konsistent. In der Beschriftung der Schaltfläche wird er als „Schlüsselbund“ bezeichnet. In den Optionen wird er als „Schlüsselspeicher“ bezeichnet.

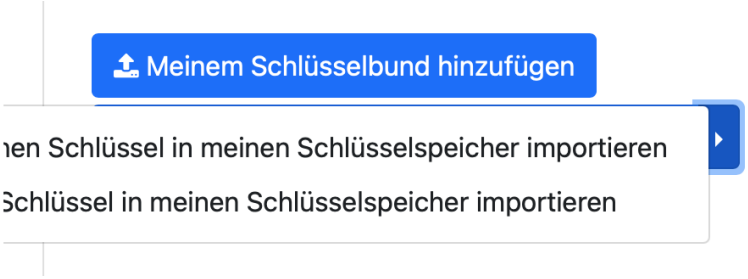

Abbildung 6: Optionen für „Separat meinem Schlüsselbund hinzufügen“ in [19]

Mithilfe der Schaltfläche „Eingaben zurücksetzen“ können schließlich die Eingabefelder für p, q, e sowie „Name“ in ihren leeren Ausgangszustand zurückgesetzt werden.

3.1.2. Tab „Verschlüsselung“

Unter dem Tab „Verschlüsselung“ (für eine Gesamtansicht siehe Abbildung A.2 auf Seite 60) kann zunächst zwischen dem Modus „Verschlüsseln“ und „Entschlüsseln“ ausgewählt werden. Anschließend wird ein öffentlicher bzw. privater Schlüssel aus den gespeicherten Schlüsseln ausgewählt.

Darauf folgen einige Optionen für die Codierung bzw. Decodierung zwischen Klartext und Klartextrepräsentanten:

- Die Option „Alphabet“ ermöglicht eine Auswahl zwischen ASCII-256 oder einem selbstdefinierten Alphabet.

- Unter „Trennung“ kann zwischen Komma, Leerzeichen und Hashtag als Trennzeichen gewählt werden.

- Unter „Umwandlungsmethode“ stehen die Optionen „b-adisch“ und „Basensystem (Verkettung)“ zur Auswahl. Beide Optionen stehen für eine b-adische Darstellung gemäß Gleichung (11) auf Seite 15. Dabei entspricht die Basis b bei der Option „b-adisch“ der Alphabetlänge l und bei der Option „Basensystem (Verkettung)“ der kleinstmöglichen Zehnerpotenz, sodass $b \geq l$.

- Die Option „Blocklänge“ definiert die Anzahl der Stellen k in der b-adischen Darstellung gemäß Gleichungen (11) und (12) auf Seite 15 und auf Seite 16.

Wählt man unter „Umwandlungsmethode“ die Option „Basensystem (Verkettung)“, stehen auch Blocklängen zur Auswahl, die zu Klartextrepräsentanten $m \geq n$ führen (siehe Abbildung A.4 auf Seite 62). Für derartige Klartextrepräsentaten sind die Ver- und Entschlüsselungsfunktionen keine inversen Abbildungen (siehe Gleichung (6) auf Seite 13).

In der nachfolgenden Sektion erfolgt die Anpassung weiterer Optionen:

- Unter „Eingabetyp“ kann zwischen „Text“ und „Zahlen“ ausgewählt werden.

- Die Option „Zahlensystem“ ermöglicht die Ein- und Ausgabe von Zahlen im dezimalen, oktalen, binären oder hexadezimalen Zahlensystem.

Über dem Eingabefeld selbst kann wiederum ausgewählt werden, ob die Eingabe manuell oder in Form einer Datei erfolgen soll.

Drei Ausgabefelder geben abhängig von den gewählten Optionen die Unterteilung der Eingabe in Blöcke, deren numerische Darstellung zur gewählten Basis (gemäß Option „Zahlensystem“) sowie den Geheim- bzw. Klartext oder die Geheimtext- bzw. Klartextrepräsentanten aus.

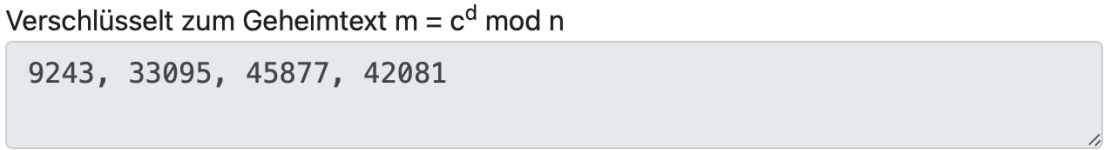

Abbildung 7: Beschriftung im Modus „Entschlüsseln“ in [19]

Im Modus „Entschlüsseln“ wird mit „Verschlüsselung zu Geheimtext …“ eine fehlerhafte Beschriftung über dem Ausgabefeld der Klartextrepräsentanten angezeigt (siehe Abbildung 7).

3.1.3. Tab „Schlüssel“

Unter dem Tab „Schlüssel“ (für eine Gesamtansicht siehe Abbildung A.3 auf Seite 61) können gespeicherte Schlüssel anhand einer tabellarischen Ansicht verwaltet werden. Im Ausgangszustand ist in dieser Tabelle bereits ein Beispielschlüssel unter dem Namen „Alice“ hinterlegt.

Über entsprechende Schaltflächen in der Tabelle können private bzw. öffentliche Schlüssel separat exportiert oder gelöscht werden. Es besteht zudem die Möglichkeit, mehrere Schlüssel in der Tabelle auszuwählen und diese mithilfe entsprechender Schaltflächen gemeinsam zu exportieren oder zu löschen.

Die Schlüssel werden in Form einer JSON-Datei (`.json`) exportiert. Codebeispiel 16 zeigt die entsprechende Objektstruktur.

```
[
  {
    name: "Alice",
    keytype: "private",
    keys: { d: "17791", n: "55189" }
  },
  {
    name: "Alice",
    keytype: "public",
    keys: { e: "7231", n: "55189" }
  }
]
```

Codebeispiel 16: Objektstruktur für den Schlüsselexport als JSON-Datei in [19]

Eine weitere Schaltfläche ermöglicht das Importieren von Schlüsseldateien aus dem lokalen Dateisystem. Hierbei fällt auf, dass beim Import lediglich geprüft wird, ob die Datei eine gültige JSON-Datei ist. Eine Validierung der eigentlichen Schlüsseldaten bleibt aus. Somit können auch ungültige oder unvollständige Schlüsseldaten importiert werden (siehe Abbildung 8).

Ungültige Schlüsseldaten werden unter dem Tab „Verschlüsseln“ zwar als solche erkannt, die dazugehörige Fehlermeldung wird jedoch auf den meisten Bildschirmen außerhalb des Bildausschnitts angezeigt. Sie erscheint für wenige Sekunden am oberen Ende des

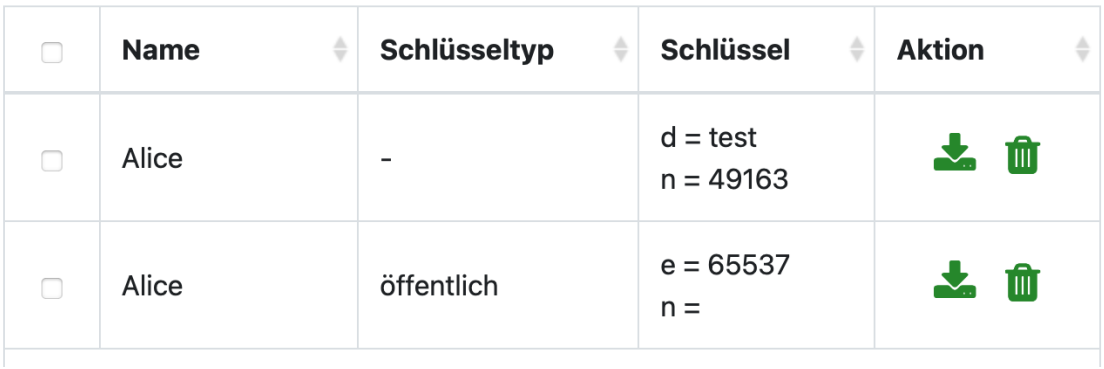

Abbildung 8: Import ungültiger Schlüsseldaten in [19]

Tabs – und zwar erst, nachdem Lernende heruntergescrollt und mit der Schaltfläche „Verschlüsseln“ bzw. „Entschlüsseln“ am unteren Ende des Tabs interagiert haben.

3.2. Anforderungskatalog

Aus den Beobachtungen des Status quo werden nachfolgend entsprechende Anforderungen für die Implementierung der neuen Anwendung abgeleitet. Die Grundfunktionalität der bestehenden Anwendung funktioniert bis auf einige wenige Fehler gut und bedarf keiner großen Änderungen. Lediglich in didaktischer Hinsicht ergeben sich zahlreiche Verbesserungsmöglichkeiten.

Statt als Unteranwendung von „RSA visuell und mehr“ soll die Funktionalität in einer eigenständigen Anwendung namens „RSA didaktisch“ bereitgestellt werden. Die Unterteilung in drei Tabs ist als positiv zu bewerten und ermöglicht eine sequentielle Erarbeitung der drei Konzepte Schlüsselgenerierung, Schlüsselverwaltung sowie Ver- und Entschlüsselung. Um die Funktionalität der Tabs jedoch eindeutiger in ihrer Benennung widerzuspiegeln, sollen sie gemäß Tabelle 6 umbenannt werden.

RSA visuell und mehr	RSA didaktisch
Schlüsselgenerierung	Schlüssel generieren
Verschlüsselung	Ver- und Entschlüsseln
Schlüssel	Schlüssel verwalten

Tabelle 6: Benennung der Tabs in „RSA visuell und mehr“ und „RSA didaktisch“

3.2.1. Tab „Schlüssel generieren“

Unter dem Tab „Schlüssel generieren“ soll in der neuen Anwendung folgende Funktionalität bereitgestellt werden:

- Lernende sollen ein Schlüsselpaar automatisch generieren können.
- Lernende sollen die Parameter für das automatische Generieren eines Schlüsselpaares anpassen können.
- Lernende sollen ein Schlüsselpaar durch manuelle Eingaben generieren können.
- Lernende sollen ein generiertes Schlüsselpaar zur Verwendung in den weiteren Tabs abspeichern können.

Dabei soll auch der in Kapitel 3.1.1 beschriebene Fehler behoben werden, sodass beim automatischen Generieren eines Schlüsselpaares unter keinen Umständen eine Zeitüberschreitung im Browser auftritt.

Didaktische Evaluierung In didaktischer Hinsicht ergeben sich für den Tab „Schlüssel generieren“ folgende Verbesserungsmöglichkeiten:

- Der private Exponent wird lediglich in einer separaten Zusammenfassung aufgeführt. Es ist für Lernende nicht ersichtlich, wie sich der private Exponent berechnet bzw. wie er mit dem öffentlichen Exponenten im Zusammenhang steht. Auch wird nicht ersichtlich, wie sich die Wahl zwischen $\phi(n)$ und $\lambda(n)$ auf den privaten Exponenten auswirkt. Hier kann der private Exponent als weiteres Eingabefeld implementiert werden, sodass Zusammenhänge zwischen den Schlüsselparametern besser visualisiert werden können.
- Bezeichnungen sollen innerhalb der gesamten Anwendung konsistent sein. Ferner soll auch auf eine Übereinstimmung mit bereits bestehenden Anwendungen in CTO, wie etwa „RSA (Schritt für Schritt erklärt)“, geachtet werden. Dort wird explizit die alternative Schlüsselgenerierungen anhand von $\lambda(n)$ eingeführt. Sie soll in „RSA didaktisch“ also nicht unter der Bezeichnung „kgV“ aufgeführt werden.
- Um die Schlüsselgenerierung direkt anhand von Beispielwerten zu verdeutlichen, sollen alle Eingabefelder beim ersten Aufrufen der Seite bereits mit einem Lösungsbeispiel befüllt sein. Darüber hinaus können einige Änderungen vorgenommen werden, um das Lernen anhand von Lösungsbeispielen stärker zu unterstützen. Lernende sollen nicht nur den öffentlichen Exponenten e separat generieren können,

sondern auch die Primfaktoren p und q. So können Lösungsbeispiele Schritt für Schritt generiert und nachvollzogen werden.

- Beim automatischen Generieren eines Schlüsselpaares kann das Intervall für die möglichen Werte der Primfaktoren p und q anhand eines dezimalen Minimums bzw. Maximums angegeben werden. Um Lernende dafür zu sensibilisieren, dass ein Schlüsselpaar in realen RSA-Anwendungen unter Angabe einer Schlüssellänge in bit erfolgt, soll dies als weitere Option angeboten werden.

- Die Option, den privaten bzw. den öffentlichen Schlüssel separat abzuspeichern soll entfernt werden. Zum einen können private und öffentliche Schlüssel unter dem dafür vorgesehenen Tab „Schlüssel verwalten“ separat verwaltet werden. Zum anderen birgt diese Option das Risiko, dass inkompatible Schlüsselpaare abgespeichert werden (das heißt, dass private und öffentliche Schlüssel unter demselben Namen abgespeichert werden, obwohl diese nicht zueinander passen). Das Eingabefeld für den Namen des Schlüsselpaares ist für die unmittelbare Schlüsselgenerierung zudem nicht relevant und soll erst angezeigt werden, sobald Lernende ein Schlüsselpaar speichern möchten.

- Das direkte Feedback bei der Validierung von Eingaben ist durchaus positiv zu bewerten. Da dieses Feedback jedoch nach wenigen Sekunden verschwindet, kann es leicht übersehen werden. Ist es einmal verschwunden, können Lernende nicht mehr nachvollziehen, warum eine Eingabe ungültig ist. Fehlermeldungen sollen persistent angezeigt werden und gezielte Hilfestellungen dazu anbieten, wie Lernende eine gültige Eingabe erreichen können.

  Die folgenden Fehlerzustände sollen behandelt werden:

  - p, q, e, d nicht definiert
  - $p, q, e, d \notin \mathbb{N}$
  - $p, q, e < 3$
  - $p, q \notin \mathbb{P}$
  - $\text{ggT}(e, \phi(n)) \neq 1$ bzw. $\text{ggT}(e, \lambda(n)) \neq 1$
  - $e, d \geq \phi(n)$ bzw. $e, d \geq \lambda(n)$
  - $e \cdot d \not\equiv 1 \bmod \phi(n)$ bzw. $e \cdot d \not\equiv 1 \bmod \lambda(n)$

3.2.2. Tab „Ver- und Entschlüsseln“

Unter dem Tab „Ver- und Entschlüsseln“ soll in der neuen Anwendung folgende Funktionalität bereitgestellt werden:

- Lernende sollen einen öffentlichen bzw. privaten Schlüssel zum Ver- bzw. Entschlüsseln auswählen können.
- Lernende sollen Klartextrepräsentanten unter Einsatz eines öffentlichen Schlüssels zu Geheimtextrepräsentanten verschlüsseln können.
- Lernende sollen Geheimtextrepräsentanten unter Einsatz eines privaten Schlüssels zu Klartextrepräsentanten entschlüsseln können.
- Lernende sollen einen Klartext zu Klartextrepräsentanten codieren und dabei die Parameter der Codierung anpassen können.
- Lernende sollen Klartextrepräsentant zu einem Klartext decodieren und dabei die Parameter der Decodierung anpassen können.
- Lernende sollen die Ausgabe der Ver- bzw. Entschlüsselung als Textdatei herunterladen können, um diese mit anderen Lernenden zu teilen.
- Lernende sollen die Eingabe der Ver- bzw. Entschlüsselung als Textdatei importieren können.

Die bei der Aus- bzw. Eingabe zu verwendenden Textdateien (`.txt`) können regulär über das jeweilige Dateisystem des Endgeräts verwaltet und geteilt werden.

Werden Geheimtextrepräsentanten als Textdatei heruntergeladen, enthalten diese keine Informationen über die zugrundeliegenden Parameter einer womöglich vorangegangenen Codierung von Klartext zu Klartextrepräsentanten. Um aus den Geheimtextrepräsentanten nach einer Entschlüsselung zu Klartextrepräsentanten letztere wieder zu einem Klartext zu decodieren, müssen Lernende die zugrundeliegenden Parameter der Codierung außerhalb der Anwendung kommunizieren. Diese Parameter sind Alphabet, Codierung und Blocklänge.

Darüber hinaus soll auch der in Kapitel 3.1.2 beschriebene Fehler bei der Berechnung der maximalen Blocklänge behoben werden, sodass bei der Codierung eines Klartexts niemals Klartextrepräsentanten $m \geq n$ erzeugt werden.

Didaktische Evaluierung In didaktischer Hinsicht ergeben sich für den Tab „Ver- und Entschlüsseln“ folgende Verbesserungsmöglichkeiten:

- Ein Verständnis der Textcodierung bzw. -decodierung ist zum wesentlichen Verständnis der Ver- und Entschlüsselung im RSA-Kryptosystem zunächst nicht notwendig. Diese beiden Prozesse können also sequentiell präsentiert werden. Dazu soll der monolithische Aufbau der bestehenden Anwendung in zwei separate Abschnitte aufgespalten werden. In einem Abschnitt wird dabei nur die Ver- bzw. Entschlüsselung anhand von Zahlen gezeigt. In dem anderen Abschnitt wird die Codierung von Klartext zu Klartextrepräsentanten bzw. die Decodierung von Geheimtextrepräsentanten zu Geheimtext demonstriert.

- Zur besseren Nachvollziehbarkeit sollen für die Textcodierung bzw. -decodierung nicht nur Endergebnisse angezeigt, sondern auch Zwischenschritte visualisiert werden. So können Lernende das Verfahren anhand der einzelnen Lösungsschritte nachvollziehen. Durch eine dynamische Anpassung der Lösungsschritte bei Änderung der Codierungs- bzw. Decodierungsparameter (wie z. B. der Blocklänge) kann zudem das Verständnis der jeweiligen Zusammenhänge gefestigt werden.

- Bei der Ver- bzw. Entschlüsselung sollen nur jene Optionen angezeigt werden, die für den jeweiligen Kontext relevant sind. Werden beispielsweise Klartextrepräsentanten verschlüsselt, müssen keine Optionen für die Textcodierung angezeigt werden. Ebenso müssen bei der Entschlüsselung keine Optionen für die Eingabe von Text angezeigt werden. Die Entschlüsselung erfolgt immer auf Grundlage von Geheimtextrepräsentanten. Um die UI auf das Wesentliche zu reduzieren sollen diese Optionen dynamisch auf Grundlage der jeweils ausgewählten Optionen ein- bzw. ausgeblendet werden.

- ASCII-256 als Alphabetoption ist für Lernende intransparent. Zum einen existiert kein einheitlicher Standard für ASCII-256. Lernende müssten also zunächst herausfinden, welche Alphabetzuordnung konkret implementiert wurde (im Falle der bestehenden Anwendung basiert diese auf der Zeichencodierung gemäß Windows-1252). Zum anderen enthält dieses Alphabet an den Stellen 0 bis 31 nichtdruckbare Zeichen. Die eigentliche Zuordnung von Alphabetzeichen zu Zahlen beginnt also erst bei 32, was für Lernende ohne das nötige Vorwissen wahllos wirken kann. Diese Option soll zugunsten der manuellen Alphabetauswahl gestrichen werden. Durch die zusätzliche Implementierung einer Zuordnungstabelle kann die Alphabetzuordnung für Lernende nachvollziehbarer gestaltet werden.

  Auch die Option, Klartext- bzw. Geheimtextrepräsentanten in verschiedenen Zahlensystemen darzustellen, trägt nichts zum wesentlichen Verständnis des RSA-Kryptosystems bei und soll entfernt werden.

- Darüber hinaus soll sichergestellt werden, dass alle Optionen und Eingabefelder konsistent beschriftet sind. Insbesondere soll klar zwischen Klartext und Klartextrepräsentanten bzw. Geheimtextrepräsentanten unterschieden werden.

- Wie auch bei der Schlüsselgenerierung sollen auch unter diesem Tab alle Fehlerzustände durch persistente Fehler- bzw. Warnmeldungen angezeigt werden.

  Die folgenden Fehlerzustände sollen behandelt werden:

  - Klartext-/Geheimtextrepräsentanten $m_i, c_i \geq n$
  - Klartext-/Geheimtextrepräsentanten $m_i, c_i \notin \mathbb{N}_0$
  - Alphabetlänge $l > n$
  - Alphabetlänge $l < 2$
  - Klartext enthält Zeichen, die nicht im Alphabet enthalten sind
  - Zeichenlänge des Klartexts ist kein Vielfaches der Blocklänge

3.2.3. Tab „Schlüssel verwalten“

Unter dem Tab „Schlüssel verwalten“ soll in der neuen Anwendung folgende Funktionalität bereitgestellt werden:

- Lernende sollen gespeicherte bzw. importierte Schlüssel in einer tabellarischen Ansicht einsehen können.

- Lernende sollen gespeicherte Schlüssel exportieren können, um diese persistent auf dem lokalen Dateisystem zu sichern.

- Lernende sollen öffentliche Schlüssel exportieren können, um diese mit anderen Lernenden zu teilen.

- Lernende sollen öffentliche Schlüssel importieren können, um damit Nachrichten für andere Lernende zu verschlüsseln.

- Lernende sollen gespeicherte bzw. importierte Schlüssel aus der tabellarischen Ansicht löschen können.

Die Schlüssel werden im JSON-Format exportiert und importiert. Die Kompatibilität mit der bestehenden Anwendung wird durch Verwendung der gleichen Objektstruktur sichergestellt (siehe Codebeispiel 16 auf Seite 34). JSON-Dateien können auf gängigen Mobil- und Desktopgeräten problemlos geöffnet und betrachtet werden. Zum Teilen können die Mechanismen der jeweiligen Endgeräte verwendet werden.

Didaktische Evaluierung In didaktischer Hinsicht ergeben sich für den Tab „Schlüssel verwalten“ folgende Verbesserungsmöglichkeiten:

- In der bestehenden Anwendung wird keinerlei Validierung der importierten Schlüssel vorgenommen. Zwar werden ungültige Schlüssel unter dem Tab „Verschlüsseln“ als solche erkannt, um Verwirrung vorzubeugen soll diese Fehlermeldung jedoch bereits beim Import der Schlüsseldateien erfolgen.

- Die UI der tabellarischen Dateiverwaltung soll sich an bestehenden Anwendungen in CTO orientieren, um eine einheitliche Nutzererfahrung zu garantieren. Eine ähnliche Dateiverwaltung findet sich in der Anwendung „OpenSSL“ unter dem Tab „Dateien“.

4. Implementierung

Kapitel 4.1 skizziert zunächst den Projektaufbau von CTO im Allgemeinen sowie der neuen Anwendung „RSA didaktisch“ im Speziellen. Anschließend rücken in Kapitel 4.2 die während der Implementierung aufgetretenen Herausforderungen und deren Lösungsansätze in den Fokus. Abschließend behandelt Kapitel 4.3 die im Quellcode der bestehenden Anwendung identifizierten Fehler sowie deren Behebung in der neuen Anwendung.

4.1. Projektaufbau

CTO zeichnet sich durch einen modularen Aufbau aus. Die einzelnen Anwendungen sind als React-Komponenten im Ordner „ctoapps“ implementiert (siehe Abbildung 9).

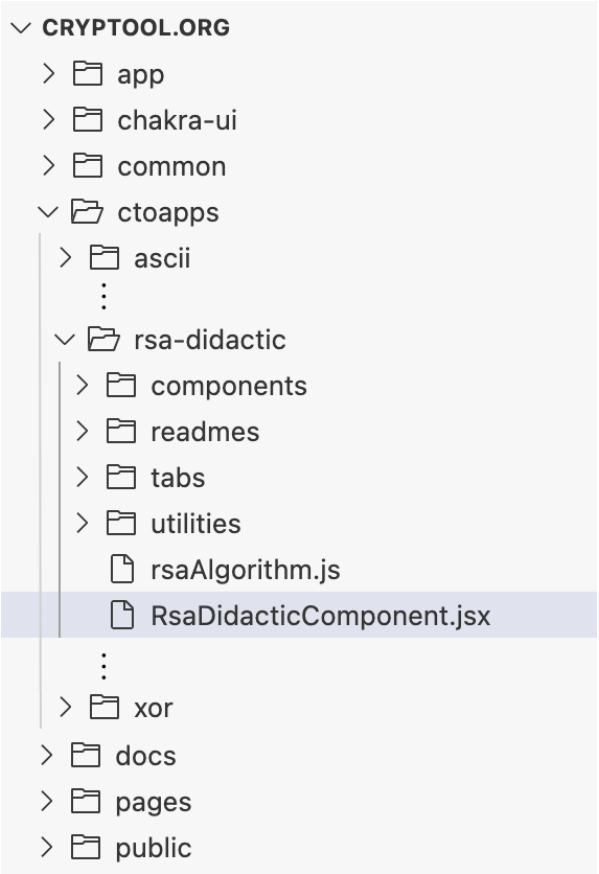

Abbildung 9: Projektstruktur von CTO

Für die Anwendung „RSA didaktisch“ wird im Ordner „ctoapps“ ein Unterordner „rsa-didactic“ angelegt. In diesem befindet sich die Wurzelkomponente „RsaDidacticComponent.jsx“. Diese bildet den Ausgangspunkt für die Anwendung.

Der Ordner „tabs“ enthält die React-Komponenten für die drei Tabs der Anwendung. Ihre Einbindung erfolgt entsprechend der in Abbildung 10 dargestellten Komponentenhierarchie ausgehend von der Wurzelkomponente.

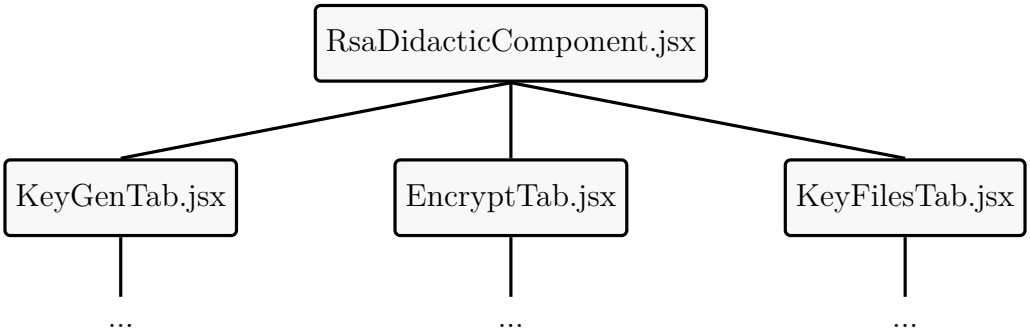

Abbildung 10: Komponentendiagramm für „RSA didaktisch“

Die Datei „rsaAlgorithm.js“ im Ordner „rsa-didactic“ enthält eine „Textbook“-Implementierung des RSA-Kryptosystems. Diese basiert im Kern auf dem Quellcode der bestehenden Anwendung [20], wobei dieser konsequenter kommentiert und mehrere Fehler behoben werden (siehe Kapitel 4.3.1). Die Datei stellt Funktionen zur Schlüsselgenerierung und -validierung sowie zur Ver- und Entschlüsselung bereit. Im Ordner „components“ befinden sich wiederverwendbare React-Komponenten, die in mehreren Tabs der Anwendung zum Einsatz kommen. Der Ordner „utilities“ enthält Hilfsfunktionen, wie z. B. „visualizeWhitespace.js“ zur Visualisierung von Leerzeichen in Zeichenfolgen.

Abbildung A.5 auf Seite 63 zeigt eine Übersicht über den Tab „Schlüssel generieren“. Abbildung A.6 auf Seite 64 zeigt den Tab „Ver- & Entschlüsseln“ im Zustand der Klartextrepräsentanteneingabe. In diesem Zustand werden die Optionen zur Textcodierung ausgeblendet. Abbildung A.8 auf Seite 66 zeigt die zusätzlichen Optionen sowie die Visualisierung der Textcodierung im Zustand der Klartexteingabe. Abbildung A.9 auf Seite 67 zeigt die Visualisierung der Textdecodierung in Richtung „Entschlüsseln“. Abbildung A.7 auf Seite 65 zeigt den Tab „Schlüssel verwalten“ mit der tabellarischen Ansicht zur Dateiverwaltung.

4.2. Herausforderungen

Im Folgenden werden zentrale Herausforderungen bei der Implementierung der neuen Anwendung sowie geeignete Lösungsansätze diskutiert.

4.2.1. Schlüssellänge

Um das Generieren eines Schlüsselpaares anhand einer vorgegebenen Schlüssellänge zu realisieren, muss das Intervall von möglichen Werten für die Primfaktoren p und q gemäß Gleichung (13) auf Seite 17 berechnet werden.

Ein Problem dieser Berechnung ist jedoch die Quadratwurzel in der Untergrenze des Intervalls. Die für dieses Projekt verwendete Bibliothek BigInteger.js sieht keine gemischte Verarbeitung von großen Ganzzahlen und Gleitkommazahlen vor. Auch der in ECMAScript spezifizierte primitive Datentyp `bigint` erlaubt keine gemischte Verarbeitung von großen Ganzzahlen und Gleitkommazahlen. Zur Berechnung der Quadratwurzel kann also nicht das in ECMAScript spezifizierte `Math`-Objekt verwendet werden, das unter `Math.SQRT2` den Wert für $\sqrt{2}$ als Gleitkommazahl bereitstellt [30, S. 466].

Aus diesem Grund kommt in der Implementierung die Approximation aus Gleichung (14) auf Seite 18 zum Einsatz. Codebeispiel 17 zeigt, wie eine Bitlänge in ein dezimales Intervall zum Generieren von p und q übersetzt wird. Dabei wird eine Bitlänge von 8 gesondert behandelt, denn die Approximation würde in diesem Fall ein Intervall ergeben, das keine zwei Primzahlen enthält.

```
92  const intervalFromBitLength = (bitLength) => {
93    // ensures primes p,q have a length of bitLength/2 and have the
        ↪ two highest
94    // order bits set to 1
95    // handles an edge case where a bitLength of 8 would otherwise
        ↪ result in an
96    // interval that contains less than 2 primes
97    const min = bitLength === 8 ? 11 : bigInt(3).shiftLeft(bitLength
        ↪ / 2 - 2)
98    const max = bigInt.fromArray(Array(bitLength / 2).fill(1), 2)
99    return { min: min.toString(), max: max.toString() }
100 }
```

Codebeispiel 17: rsa-didactic/.../GenerateKeyPairButton.jsx, Zeile 92–100

In der UI erfolgt die Auswahl der Bitlänge anhand eines Schiebereglers. Dazu werden die linearen Werte w des Schiebereglers als Bitlängen der Form 2^w aufgefasst. Damit unter dem Schieberegler die richtigen Beschriftungen angezeigt werden, müssen diese entsprechend umgerechnet werden. Codebeispiel 18 zeigt diese Berechnung in einer wiederverwendbaren Komponente, die als Props eine minimale und eine maximale Bitlänge erhält. In Zeile 31 wird zunächst ein Array erstellt, das die ganzzahligen Werte von `minSliderValue` bis `maxSliderValue` enthält. Die Methode `map()` in Zeile 34 gibt schließlich ein neues Array zurück, das Objekte der Form `{value: v, label 2 ** v}` enthält – also den Werten entsprechende Zweierpotenzen zuordnet.

```
30 const marks = useMemo(() => {
31   return Array.from(
32     { length: maxSliderValue - minSliderValue + 1 },
33     (_, i) => i + minSliderValue
34   ).map((v) => ({ value: v, label: 2 ** v }))
35 }, [minSliderValue, maxSliderValue])
```

Codebeispiel 18: rsa-didactic/.../BitLengthSlider.jsx, Zeile 30–35

Die Komponente `BitLengthSlider` wird schließlich wie in Codebeispiel 19 verwendet. Die Umrechnung von Bitlängen zu linearen Werten des Schiebereglers erfolgt innerhalb der Komponente mithilfe der Methode `Math.log2(x)` [30, S. 472].

```
1 <BitLengthSlider min={8} max={512} />
```

Codebeispiel 19: Verwendung der Komponente `BitLengthSlider`

Abbildung 11 zeigt das Endergebnis in der Anwendung. Aus didaktischen Gründen wird die Schlüssellänge auf 512 bit beschränkt. Mit zunehmender Schlüssellänge erschwert die schiere Größenordnung der involvierten Zahlen eine nachvollziehbare Visualisierung, insbesondere der Textcodierung bzw. -decodierung.

Abbildung 11: Schieberegler für die Schlüssellänge in „RSA didaktisch“

4.2.2. Langzahldarstellung in Eingabefeldern

Eine weitere Herausforderung besteht darin, die potenziell sehr langen Zahlen in den Eingabefeldern der Schlüsselparameter vor allem auch auf kleineren Bildschirmen darzustellen. Dies gilt im Besonderen für mobile Endgeräte, da dort ein horizontales Scrollen in Eingabefeldern nur umständlich möglich ist.

Ein möglicher Lösungsansatz ist die Ersetzung der Eingabefelder (`Input`) durch scrollbare Textbereiche (`Textarea`) (siehe Abbildung 12). Dies erweist sich jedoch optisch als nicht zufriedenstellend, da die Eingabewerte aufgrund der Scrollleisten nicht mehr in einer Linie mit den Beschriftungen der Eingabefelder ausgerichtet sind. Eine weitere Möglichkeit ist die komplette Neugestaltung der UI, sodass Beschriftung, Schaltfläche und Textbereich nicht in einer Reihe, sondern vertikal übereinander angeordnet sind.

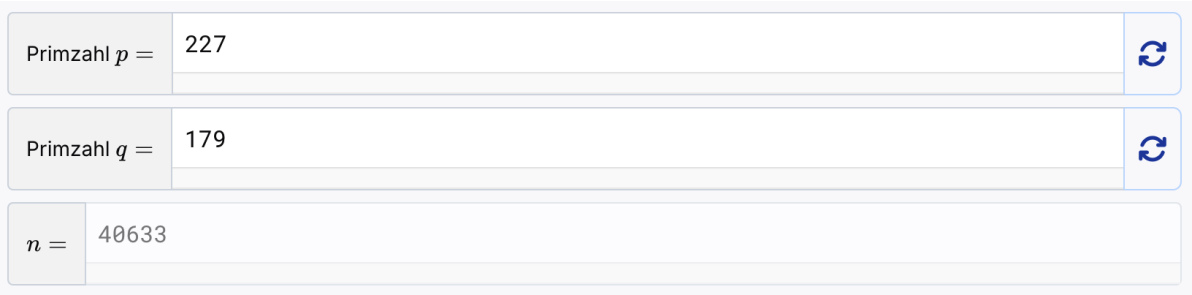

Abbildung 12: Scrollleisten in den Eingabefeldern in „RSA didaktisch“

Um die UI jedoch nicht unnötig in die Länge zu ziehen – vor allem im Hinblick auf mobile Endgeräte – wird ein dritter Lösungsansatz verfolgt. Dabei werden die Eingabewerte derart gekürzt, dass sie ohne Überlauf in das Eingabefeld passen. Der Mittelteil der Eingabewerte wird durch ein Auslassungszeichen (…) ersetzt. Wird das Eingabefeld fokussiert (über eine Tastatur-, Maus- oder Touch-Interaktion), ersetzt der vollständige Wert den gekürzten Wert. Dies ermöglicht es, sehr lange Zahlen anhand ihrer ersten bzw. letzten Ziffern miteinander zu vergleichen, während weiterhin normal mit dem Eingabefeld interagiert werden kann (z. B. um den Eingabewert anzupassen oder den vollständigen Eingabewert zu kopieren). Abbildung 13 illustriert dieses Verhalten anhand eines Zustandsübergangsdiagramms.

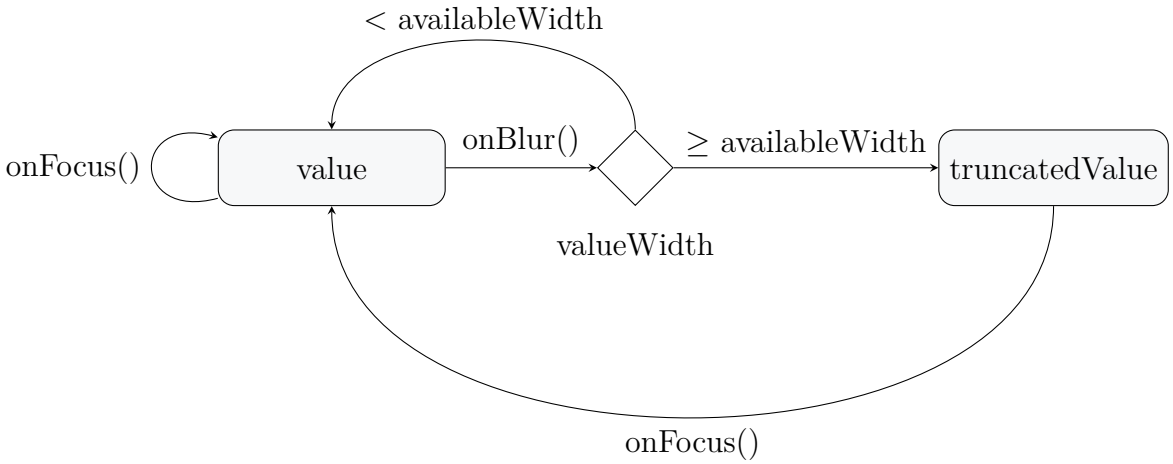

Abbildung 13: Zustandsübergangsdiagramm zur Darstellung gekürzter Eingabewerte

Um diesen Lösungsansatz zu implementieren, muss zunächst die von den Eingabewerten vereinnahmte Breite in Pixeln bestimmt werden. Da es sich bei der Schriftart in den Eingabefeldern um eine nichtproportionale Schriftart (engl.: *monospaced font*) handelt, kann man sich dazu eines Tricks bedienen (siehe Codebeispiel 20).

```
8  export default function measureMonospaceWidth(fontFamily, fontSize)
     ↪ {
9    const span = document.createElement("span")
10
11   span.style.fontFamily = fontFamily
12   span.style.fontSize = fontSize
13
14   span.style.position = "absolute"
15   span.style.width = "1ch"
16
17   document.body.appendChild(span)
18   const width = span.getBoundingClientRect().width
19   document.body.removeChild(span)
20
21   return width
22 }
```

Codebeispiel 20: rsa-didactic/.../measureMonospaceWidth.js, Zeile 8–22

Dabei wird in Zeile 9 ein leeres Element im DOM des Browsers erzeugt, dem die Schriftart und die Schriftgröße der Eingabefelder zugewiesen wird. Die Breite des Elements wird in Zeile 15 als `1ch` definiert. Diese Einheit entspricht genau der Breite des Zeichens `0` in der gegebenen Schriftart und Schriftgröße [21]. Da es sich bei der Schriftart um eine nichtproportionale Schriftart handelt, entspricht die Breite des Zeichens `0` der Breite aller anderen Ziffern in dieser Schriftart.

```
99  const valueWidth = charWidth * value.length
100 if (valueWidth > availableWidth) {
101   const ellipsis = "..."
102   const maxLength = Math.floor(availableWidth / charWidth) -
        ↪ ellipsis.length
103   const halfLength = maxLength / 2
104
105   return (
106     value.slice(0, Math.ceil(halfLength)) +
107     ellipsis +
108     value.slice(-Math.floor(halfLength))
109   )
110 } else {
111   return value
112 }
```

Codebeispiel 21: rsa-didactic/.../KeyGenField.jsx, Zeile 99–112

Codebeispiel 21 zeigt in Zeile 99, wie die Breite des gesamten Eingabewerts berechnet wird, indem die zuvor gemessene Breite eines Zeichens mit der Gesamtlänge des Einga-

bewerts multipliziert wird. Anschließend wird in Zeile 100 die Breite der Eingabewerte mit der verfügbaren Breite innerhalb des Eingabefeldes abgeglichen, um schließlich in Zeile 102 eine maximale Zeichenlänge `maxLength` für den Eingabewert zu ermitteln. Dabei wird die Länge des Auslassungszeichens bereits abgezogen. Der verkürzte Wert besteht dann aus den ersten `halfLength` Zeichen, dem Auslassungszeichen und den letzten `halfLength` Zeichen. Bei einem ungeraden Wert für `maxLength` wird entsprechend eine Hälfte auf- und die andere Hälfte abgerundet.

Um den gekürzten Eingabewert bei einer Interaktion mit dem Eingabefeld durch den vollständigen Eingabewert zu ersetzen, können die Events `onFocus` und `onBlur` abgefangen und der jeweilige Zustand in einer Zustandsvariable gespeichert werden. Ersteres Event wird ausgelöst, wenn das Eingabefeld über Maus-, Touch- oder Tastaturinteraktion angewählt wird. Letzteres Event wird ausgelöst, wenn das Eingabefeld wieder abgewählt wird.

```
74 const [inputFocused, setInputFocused] = useState(false)
75
76 /**
77   * Sets focused state of the input element to true, moves cursor
       ↪ to the end.
78   */
79 const handleFocus = () => {
80   setInputFocused(true)
81   inputRef.current.setSelectionRange(value.length, value.length)
82 }
83
84 /**
85   * Sets focused state of the input element to false.
86   */
87 const handleBlur = () => {
88   setInputFocused(false)
89 }
```

Codebeispiel 22: rsa-didactic/.../KeyGenField.jsx, Zeile 74–89

Damit die jeweiligen Maße bei einer Veränderung der UI (z. B. durch eine Veränderung der Fensterbreite) dynamisch neu berechnet werden, kommt der Hook `useEffect()` zum Einsatz, um mithilfe der vom Browser bereitgestellten Web API `ResizeObserver` [22] alle Veränderungen der Dimensionen des Eingabefeldes abzufangen und die verfügbare Breite des Eingabefeldes daraufhin neu zu berechnen.

Abbildung 14 zeigt das Endergebnis. Das Eingabefeld für Primzahl p zeigt den fokussierten Zustand. Die beiden anderen Eingabefelder zeigen den nichtfokussierten Zustand.

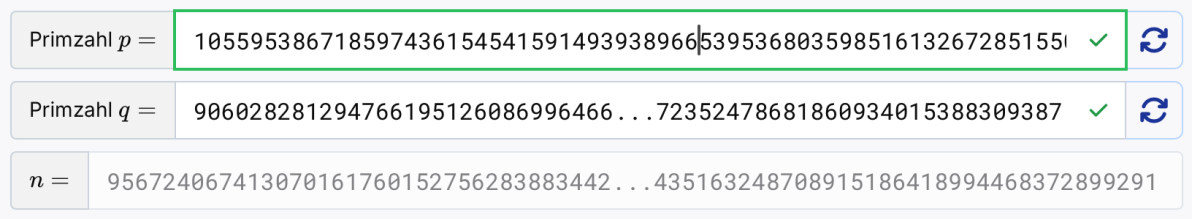

Abbildung 14: Gekürzte Werte in den Eingabefeldern in „RSA didaktisch“

4.2.3. Klartext- und Alphabetvalidierung

Im Rahmen der Klartexteingabe besteht eine Herausforderung darin, Lernende dafür zu sensibilisieren, dass nur im Alphabet enthaltene Zeichen als gültige Eingaben verwendet werden können. Daher wird direkt bei der Texteingabe durch die neu implementierte Komponente `CharAlert` darauf hingewiesen. Eine Schaltfläche bietet die Möglichkeit, die nicht enthaltenen Zeichen zum Alphabet hinzuzufügen (siehe Abbildung 15).

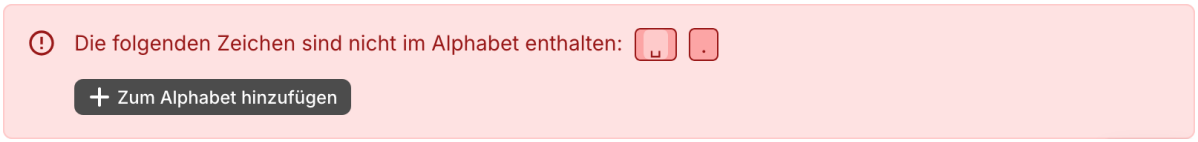

Abbildung 15: Komponente `CharAlert` in „RSA didaktisch“

Die Anpassung des Alphabets geschieht mithilfe der bereits in CTO implementierten Komponente `AlphabetOptionCard`. Diese enthält jedoch keine Funktion zur Prüfung auf Duplikate. Codebeispiel 23 zeigt die Duplikatprüfung unter Zuhilfenahme der Datenstruktur `Set` [30, S. 641]. Duplikate werden automatisch verworfen und mithilfe der wiederverwendbaren Komponente `CharAlert` kenntlich gemacht (siehe Abbildung 16).

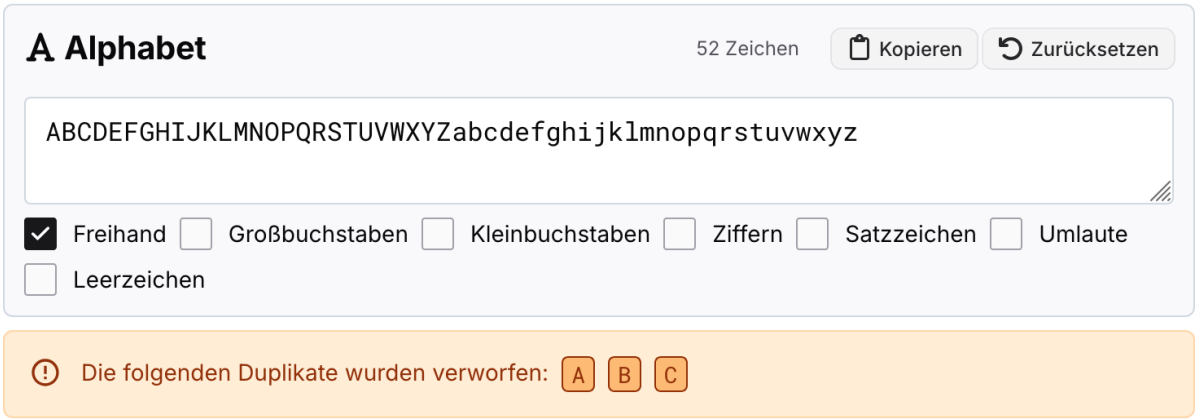

Abbildung 16: Duplikatbehandlung bei der Alphabetanpassung in „RSA didaktisch“

```
338 const handleAlphabetChange = (input) => {
339   setAlphabetDuplicates([])
340   // string input means freestyle input
341   if (typeof input === "string") {
342     const unique = new Set()
343     const duplicates = []
344     for (const e of input) {
345       if (unique.has(e)) duplicates.push(e)
346       else unique.add(e)
347     }
348     if (unique.size < input.length) {
349       setAlphabetDuplicates(duplicates)
350       input = Array.from(unique)
351     }
352   }
353   setAlphabet(input)
354 }
```

Codebeispiel 23: rsa-didactic/.../EncryptTab.jsx, Zeile 338–354

4.2.4. Visualisierung der Textcodierung

Um die Textcodierung bzw. -decodierung nachvollziehbarer zu gestalten, kommt ein Konzept zur schrittweisen Visualisierung der b-adischen Darstellung zum Tragen. Eine Grundvoraussetzung dafür ist die Zuordnung von Alphabetzeichen zu Ganzzahlen. Diese wird anhand einer Zuordnungstabelle verdeutlicht. Da auch nichtdruckbare Zeichen, wie Leerzeichen oder Zeilenumbrüche, als Teil des Alphabets vorkommen können, müssen diese entsprechend visualisiert werden.

Zuordnungstabelle: Zeichen ↔ Zahlen

␣	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19

Abbildung 17: Zuordnungstabelle in „RSA didaktisch“

Abbildung 17 zeigt die Zuordnungstabelle für ein Alphabet, das an der ersten Stelle das Leerzeichen enthält. Die Funktion `visualizeWhitespace()` ersetzt Leerzeichen, Tabulatorzeichen und Zeilenumbrüche durch eine Visualisierung auf Grundlage der in Chakra UI vordefinierten Komponente `Kbd` [12].

Die Visualisierung der b-adischen Darstellung erfolgt in vier Schritten:

1. Im ersten Schritt wird der Klartext in Blöcke der gewählten Blocklänge unterteilt.

2. Im zweiten Schritt erfolgt für jeden Block die Zuordnung von Alphabetzeichen zu Ganzzahlen.

3. Im dritten Schritt wird die b-adische Darstellung gemäß Gleichung (11) auf Seite 15 mitsamt Zwischenschritten visualisiert.

4. Im letzten Schritt werden die finalen Klartextrepräsentanten ausgegeben.

Die einzelnen Blöcke werden dabei alternierend farblich abgesetzt, um die Zusammenhänge zwischen den einzelnen Schritten kenntlich zu machen. Abbildung 18 zeigt die Visualisierung der b-adischen Darstellung in der neuen Anwendung.

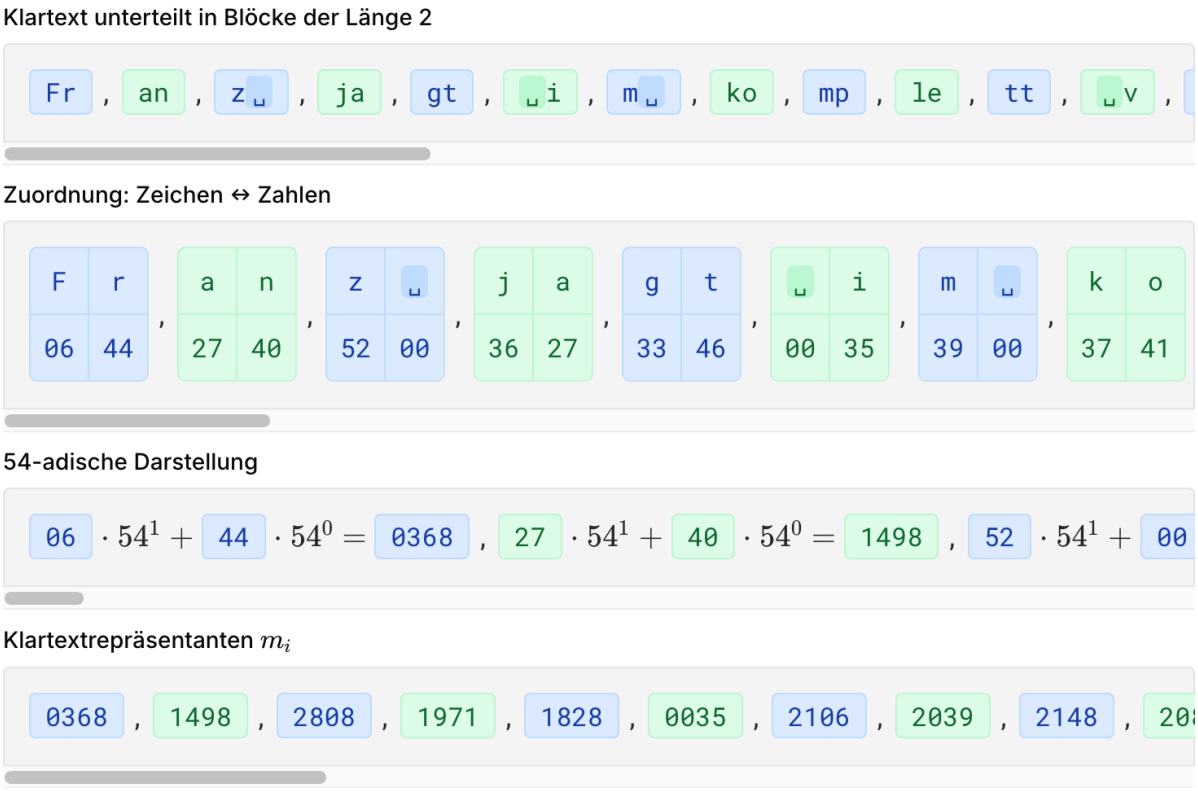

Abbildung 18: Visualisierung der b-adischen Darstellung in „RSA didaktisch“

4.3. Fehlerbehebungen

Im Folgenden werden ausgewählte Fehler im Quellcode der bestehenden Anwendung [20] beschrieben sowie Maßnahmen zu deren Behebung erläutert.

4.3.1. Schlüsselgenerierung

Wählt man in der bestehenden Anwendung ein Intervall zum Generieren der Primfaktoren p und q aus, das weniger als zwei Primzahlen enthält (beispielsweise die Werte 5 und 6 als Minimum und Maximum), führt dies zu einer Zeitüberschreitung im Browser. Auf der Suche nach dem Fehler stößt man im Quellcode der bestehenden Anwendung zunächst auf die Funktion in Codebeispiel 24.

```
70 const check_if_btw_are_minimum_two_primes = (min, max) => {
71
72     let result = []
73     let i = min
74     while (i <= max) {
75         if (bigInt(i).isPrime())
76             result.push(i)
77         i++
78         if (result.length > 2) return true
79     }
80     return bigInt(-1)
81 };
```

Codebeispiel 24: rsa-visual-and-more/.../rsaHelperFunctions.js, Zeile 70–81 [20]

Auf den ersten Blick zeigt sich, dass die Funktion – entgegen ihrer Bezeichnung – keine korrekte Prüfung auf die Existenz von mindestens zwei Primzahlen im angegebenen Intervall durchführt. Die `while`-Schleife wird in Zeile 78 erst beim Auffinden von drei Primzahlen vorzeitig verlassen. Werden in der bestehenden Anwendung z. B. die Werte 5 und 7 als Minimum und Maximum angegeben, wäre mit einer Fehlermeldung zu rechnen, was jedoch nicht der Fall ist. Der Grund dafür liegt nicht in Codebeispiel 24 selbst, sondern in der aufrufenden Funktion in Codebeispiel 25.

```
8  const get_rand_key = (min, max, lcm_hidden = true) => {
9
10   if (!check_if_btw_are_minimum_two_primes(min, max)) return bigInt
       ↪ (-1)
11   let p = bigInt.randBetween(min, max);
12   let q = bigInt.randBetween(min, max);
13   while (!p.isPrime()) {
14       p = bigInt.randBetween(min, max)
15   }
16   while (!q.isPrime() || q.equals(p)) {
17       q = bigInt.randBetween(min, max)
18   }
```

Codebeispiel 25: rsa-visual-and-more/.../rsaHelperFunctions.js, Zeile 8–18 [20]

Dort wird in der `if`-Abfrage in Zeile 10 der Rückgabewert der Funktion aus Codebeispiel 24 auf einen Wert geprüft, der als *falsy* gilt (siehe Tabelle 3 auf Seite 21). Das Problem besteht jedoch darin, dass die Funktion entweder den Wert `true` in Zeile 78 oder das Objekt `bigInt(-1)` in Zeile 80 zurückgibt, welches ebenfalls zu den *truthy* Werten zählt. Der Rückgabewert wird also in jedem Fall als `true` interpretiert, sodass niemals geprüft wird, ob das angegebene Intervall tatsächlich zwei Primzahlen enthält.

In der Folge führt die Konstellation der `while`-Schleifen im weiteren Verlauf von Codebeispiel 25 zu einer Endlosschleife in Zeile 16. Auch der Fall, dass keine Primzahl im Intervall zu finden ist (beispielsweise, wenn Minimum und Maximum gleich sind), wird nicht abgefangen und hat bereits in Zeile 13 eine Endlosschleife zur Folge.

Um das Problem zu beheben, muss also entweder die Funktion in Codebeispiel 24 einen *falsy* Wert (bzw. direkt `false`) zurückgeben oder in der aufrufenden Funktion muss explizit auf den Wert `bigInt(-1)` geprüft werden.

Unabhängig davon verbirgt sich in Codebeispiel 24 noch ein weiterer Fehler im Zusammenhang mit der impliziten Umwandlung von Datentypen in JS. In Zeile 77 wird der Iterator `i` folgendermaßen hochgezählt: `i++`. Dies erscheint zunächst unproblematisch. Als Startwert von `i` wird jedoch der Funktionsparameter `min` übernommen, welcher beim Aufruf als `bigInt` übergeben wird. Für `bigInt`-Objekte ist der Inkrementoperator `++` nicht definiert und so wird der Wert `i` in Zeile 77 zunächst in den primitiven Datentyp `number` umgewandelt, bevor er inkrementiert wird [30, S. 272].

Der primitive Datentyp `number` kann nur Ganzzahlen bis $2^{53} - 1 = 9\,007\,199\,254\,740\,991$ ohne Verlust von Genauigkeit darstellen (siehe Kapitel 2.2.1 auf Seite 22).

```
1 let a = 9007199254740991
2 a++ // a = 9007199254740992
3 a++ // a = 9007199254740992
```

Codebeispiel 26: Inkrementieren von `number` über `Number.MAX_SAFE_INTEGER` hinaus

Die Inkrementoperation in Zeile 77 von Codebeispiel 24 funktioniert also solange, bis die vom `bigInt`-Objekt repräsentierte Zahl die maximal darstellbare Ganzzahl des Datentyps `number` überschreitet. In diesem Fall wird die Zahl nicht länger korrekt inkrementiert und die `while`-Schleife kann nicht mehr terminiert werden (siehe Codebeispiel 26). Wird also ein Minimum von $2^{53} - 1 = 9\,007\,199\,254\,740\,991$ oder höher für das Intervall der Schlüsselgenerierung angegeben, resultiert dies erneut in einer Endlosschleife. Somit ist es in der bestehenden Anwendung nicht verlässlich möglich, Schlüssel zu generieren, deren Primfaktoren eine Länge von mehr als 53 bit aufweisen.

Durch Verwendung der entsprechenden Inkrementmethode `next()` des `bigInt`-Objekts wird dieses Problem in Zeile 132 von Codebeispiel 27 behoben.

```
126 const intervalHasTwoPrimes = (min, max) => {
127   let primes = 0
128   let i = bigInt(min)
129   while (i.leq(max)) {
130     if (i.isPrime()) primes++
131     if (primes >= 2) return true
132     i = i.next()
133   }
134   return false
135 }
```

Codebeispiel 27: rsa-didactic/rsaAlgorithm.js, Zeile 126–135

Zusammenfassend zeigt Codebeispiel 27 die korrigierte Version der Funktion aus Codebeispiel 24 nach Behebung aller identifizierten Fehler. Die Funktion prüft nun in Zeile 131 korrekt auf die Existenz von zwei statt drei Primzahlen im angegebenen Intervall. Außerdem gibt sie nun immer einen booleschen Wert zurück, um eine konsistentere Prüfung des Rückgabewerts zu ermöglichen. Darüber hinaus wird der Eingabewert `min` nun in Zeile 128 explizit in ein `bigInt`-Objekt umgewandelt und es werden die korrekten Inkrement- und Vergleichsmethoden verwendet (`i.next()` und `i.leq()`).

4.3.2. Berechnung der maximalen Blocklänge

Wie in Kapitel 3.1.2 beschrieben, tritt in der bestehenden Anwendung bei der Berechnung der maximalen Blocklänge für die Textcodierung bzw. -decodierung ein Fehler auf, sodass Klartextrepräsentanten $m \geq n$ entstehen können. Es wird also eine zu hohe maximale Blocklänge berechnet.

Codebeispiel 28 zeigt die entsprechende Funktion im Quellcode der bestehenden Anwendung. In der Funktion wird zur Berechnung der maximalen Blocklänge zwischen den Umwandlungsmethoden „b-adisch“ und „Basensystem (Verkettung)“ unterschieden:

1. Die Berechnung für den Fall „b-adisch“ erfolgt auf Grundlage von Gleichung (12) auf Seite 16. Jedoch kann hier nicht die Methode `Math.log(x)` des in ECMAScript spezifizierten `Math`-Objekts verwendet werden, da dies eine implizite Umwandlung der `bigInt`-Objekte zum Datentyp `number` zur Folge hätte.

2. Die Berechnung für den Fall „Basensystem (Verkettung)“ basiert auf Gleichung (11) auf Seite 15. Dabei wird ausgehend von einem jeweils maximalen m_i (das entspricht der Alphabetlänge $l - 1$) in einer Schleife solange der Exponent der Basis b hochgezählt, bis die Summe den RSA-Modul n übersteigt.

```
569 const calculate_max_blocksize = (n, encode_type, alphabet_length)
    ↪ => {
570     let expo = 1;
571     if (encode_type == 1) {// b-adisch
572         let temp = bigInt(alphabet_length).pow(bigInt(expo))
573         while (temp.lesserOrEquals(bigInt(n))) {
574             temp = (bigInt(alphabet_length).pow(bigInt(expo)))
575             if (temp.greater(n)) {
576                 return expo - 1
577             }
578             expo++
579         }
580         return expo
581     } else if (encode_type == 2) {
582         // base
583         let i = bigInt(0);
584         let w = bigInt(0);
585         alphabet_length = alphabet_length.minus(1);
586         do {
587             w = w.add((alphabet_length.minus(1)).add(bigInt(100).
                ↪ pow(i)));
588             i = bigInt(i).add(1);
589         } while (w.lesserOrEquals(n));
590         return i.minus(1);
591     }
592 }
```

Codebeispiel 28: rsa-visual-and-more/.../rsa-didactic/rsa.js, Zeile 569–592 [20]

Das Problem liegt jedoch in Zeile 587. Dort wird statt einer Multiplikation `times()` eine zweite Addition `add()` durchgeführt. Außerdem müsste hier statt `bigInt(100)` die jeweilige Zehnerpotenz basierend auf der Alphabetlänge verwendet werden. Darüber hinaus fällt auch auf, dass die Alphabetlänge bereits in Zeile 585 um 1 subtrahiert wird. Dies müsste in der Schleife also nicht noch einmal erfolgen.

Zu guter Letzt ist auffällig, dass sowohl in der ersten Schleife in Zeile 573 als auch in der zweiten Schleife in Zeile 589 die Bedingung `lesserOrEquals(bigInt(n))` geprüft wird. Auch dies führt in Randfällen zur Berechnung einer zu hohen maximalen Blocklänge und es sollte stattdessen die korrekte Bedingung `lesser(bigInt(n))` verwendet werden.

In der neuen Anwendung wird die Funktion als memoisierte Berechnung implementiert, die in Abhängigkeit von RSA-Modul, Basis und Alphabetlänge stets die korrekte maximale Blocklänge zurückgibt (siehe Codebeispiel 29).

```
451 const maxBlockLength = useMemo(() => {
452     if (selectedKey && alphabetLength >= 2) {
453       let i = bigInt(0)
454       let t = bigInt(0)
455       const max = bigInt(alphabetLength - 1)
456       do {
457         t = t.add(max.times(encodingBase.pow(i)))
458         i = i.next()
459       } while (t.lesser(selectedKey.keys.n))
460       return i.prev()
461     } else return 0
462 }, [selectedKey, alphabetLength, encodingBase])
```

Codebeispiel 29: rsa-didactic/.../EncryptTab.jsx, Zeile 451–462

4.3.3. Feedback bei ungültigen Eingaben

In der bestehenden Anwendung werden negative Eingabewerte unter dem Tab „Schlüsselgenerierung“ nicht als fehlerhaft angezeigt (siehe Kapitel 3.1.1). Die Ursache hierfür ist in Codebeispiel 30 zu finden.

```
652 if (!p.lesserOrEquals(0) && p.isPrime() && !p.equals(q)) {
653     valid_p = true;
654     $("#key-generation-p").addClass("is-valid");
655     $("#key-generation-p").removeClass("is-invalid");
656     if (input == "p") feedback_element = $("#didactic-p-is-prime");
657 } else if ((!p.lesserOrEquals(0) && !p.isPrime() && !p.equals(q))
    ↪ || p.equals(q)) {
658
659     if (input == "p") {
660         if (!p.isPrime() || p.lesserOrEquals(0)) {
661             feedback_element = $("#didactic-p-is-not-prime");
662         } else if (p.equals(q)) {
663             feedback_element = $("#p-and-q-same-input-alert");
664         }
665     }
666     valid_input = false;
```

Codebeispiel 30: rsa-visual-and-more/.../rsa-didactic/rsa.js, Zeile 652–666 [20]

Hier wird in Zeile 652 die folgende Bedingung geprüft, um eine gültige Eingabe zu erkennen:

$$\neg(p \leq 0) \land (p \in \mathbb{P}) \land \neg(p = q)$$

In Zeile 657 soll eine ungültige Eingabe anschließend folgendermaßen erkannt werden:

$$(\neg(p \leq 0) \land \neg(p \in \mathbb{P}) \land \neg(p = q)) \lor (p = q)$$

Im ersten Teil des Ausdrucks wird erneut geprüft, ob p eine positive Zahl ist. Das heißt, eine negative Zahl kann nur dann als ungültige Eingabe erkannt werden, wenn $p = q$ gilt. Dies geschieht in Zeile 660.

Nach den de-morganschen Gesetzen liefert die Negation der `if`-Bedingung in Zeile 652 genau die gewünschte Bedingung für ungültige Eingaben:

$$\neg(\neg(p \leq 0) \land (p \in \mathbb{P}) \land \neg(p = q)) = (p \leq 0) \lor \neg(p \in \mathbb{P}) \lor (p = q)$$

Dies könnte als Grundlage für die `if`-Abfrage in Zeile 657 dienen. Sofern keine weiteren Fälle, wie leere Eingaben, behandelt werden müssen, ließe sich die zweite `if`-Bedingung auch ganz weglassen, da alle Fälle bereits durch den `else`-Zweig abgedeckt werden.

In der neuen Anwendung erfolgt die Validierung von Eingabewerten zweistufig. Zunächst werden die Eingabewerte auf die Ziffern 0–9 begrenzt, sodass überhaupt keine Werte $p \notin \mathbb{N}$ sowie $p < 0$ eingegeben werden können (siehe Codebeispiel 31).

```
8  export default function removeNonNumeric(input, array = false) {
9    if (typeof input === "string" && input.length) {
10     if (array) return input.replace(/[^0-9,]/g, "").replace(/,+/g,
         ↪ ",")
11     else return input.replace(/[^0-9]/g, "")
12   } else {
13     return ""
14   }
15 }
```

Codebeispiel 31: rsa-didactic/.../removeNonNumeric.js, Zeile 8–15

Anschließend werden die Eingabewerte für eine Primzahl p mithilfe der Funktion in Codebeispiel 32 auf Fehler geprüft, wobei auch der Fall $p < 3$ bzw. $p = 2$ abgefangen wird, der in der bestehenden Anwendung als gültige Eingabe gilt.

```
143 const getPrimeError = (input) => {
144   if (input) {
145     try {
146       const p = bigInt(input)
147       if (p.lt(3)) return "isLessThanThree"
148       if (!p.isPrime()) return "isNotPrime"
149     } catch {
150       return "isInvalid"
151     }
152   } else return "isUndefined"
153 }
```

Codebeispiel 32: rsa-didactic/.../KeyGenTab.js, Zeile 143–153

5. Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine bestehende Demonstration des RSA-Kryptosystems untersucht, deren Optimierungspotenzial aufgezeigt und darauf aufbauend eine neue Anwendung entwickelt. Durch den Einsatz moderner Web-Technologien, wie React und Chakra UI, konnte eine Anwendung realisiert werden, die sich nahtlos in die aktuellste Version von CTO einfügt.

Im Zuge dessen wurden nicht nur technische Fehler im Quellcode der bestehenden Anwendung behoben, sondern auch einige didaktische Verbesserungen vorgenommen, allen voran die Visualisierung der b-adischen Darstellung. Darüber hinaus wurden Verbesserungen in der Handhabung erzielt, etwa durch eine strengere Validierung von Benutzereingaben oder die kontextbezogenere Gestaltung der Benutzeroberfläche.

Für die Zukunft bieten sich mehrere Erweiterungsmöglichkeiten an. Zum einen könnte die Anwendung um einen weiteren Tab ergänzt werden, der ein einfaches Signaturverfahren auf Grundlage von RSA demonstriert. Zum anderen könnten Lernende für einige der Sicherheitsproblematiken von „Textbook“-RSA sensibilisiert werden sowie Padding-Verfahren in die Demonstration einbezogen werden.

Schließlich könnte auch eine empirische Evaluation mit Lernenden im konkreten Unterrichtskontext wertvolle Erkenntnisse über die Wirksamkeit des didaktischen Konzepts der Anwendung liefern. Auf Grundlage dieser Erkenntnisse ließe sich die Anwendung iterativ weiter verbessern.

A. Abbildungen

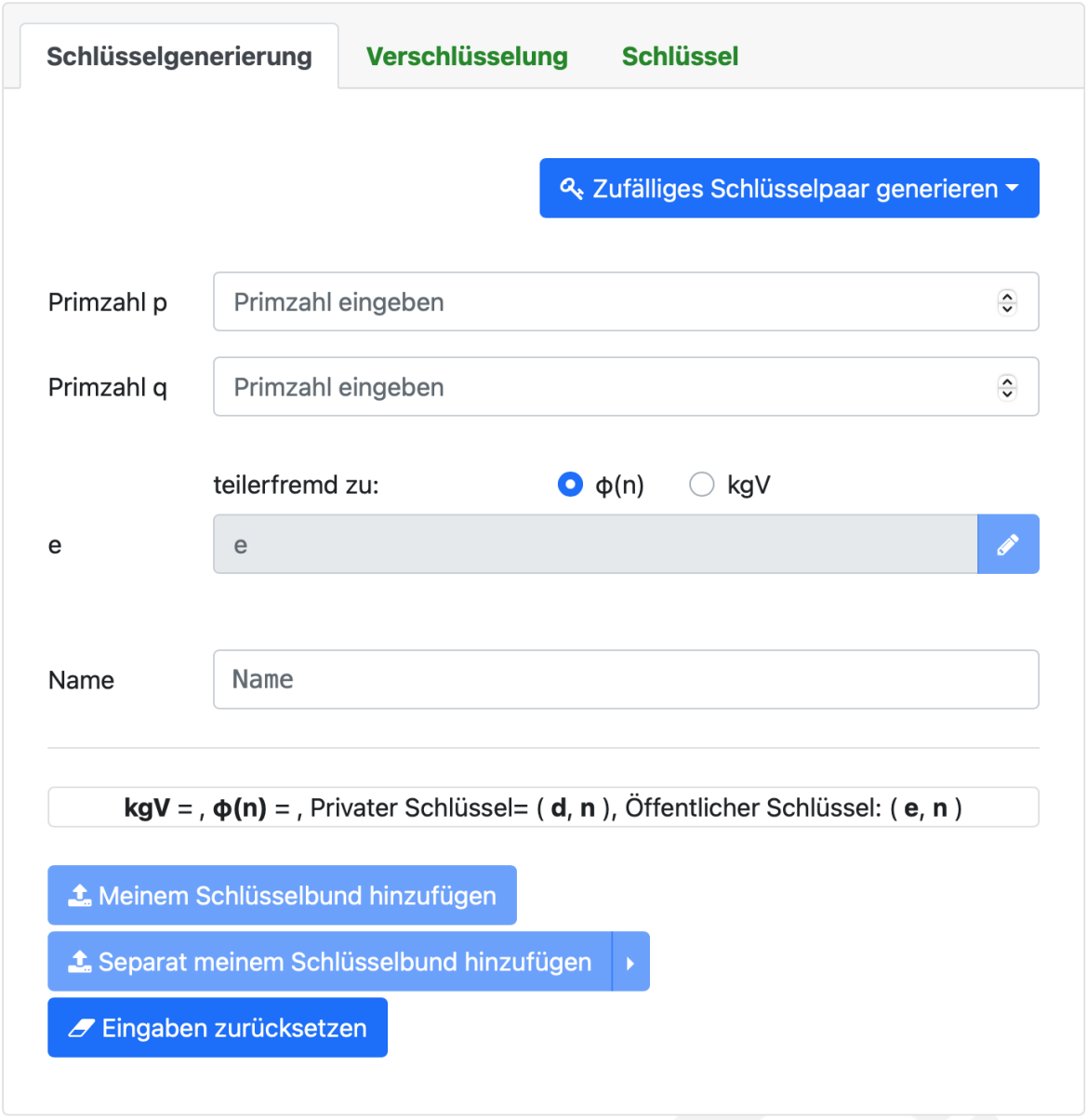

Abbildung A.1: Tab „Schlüsselgenerierung“ in [19]

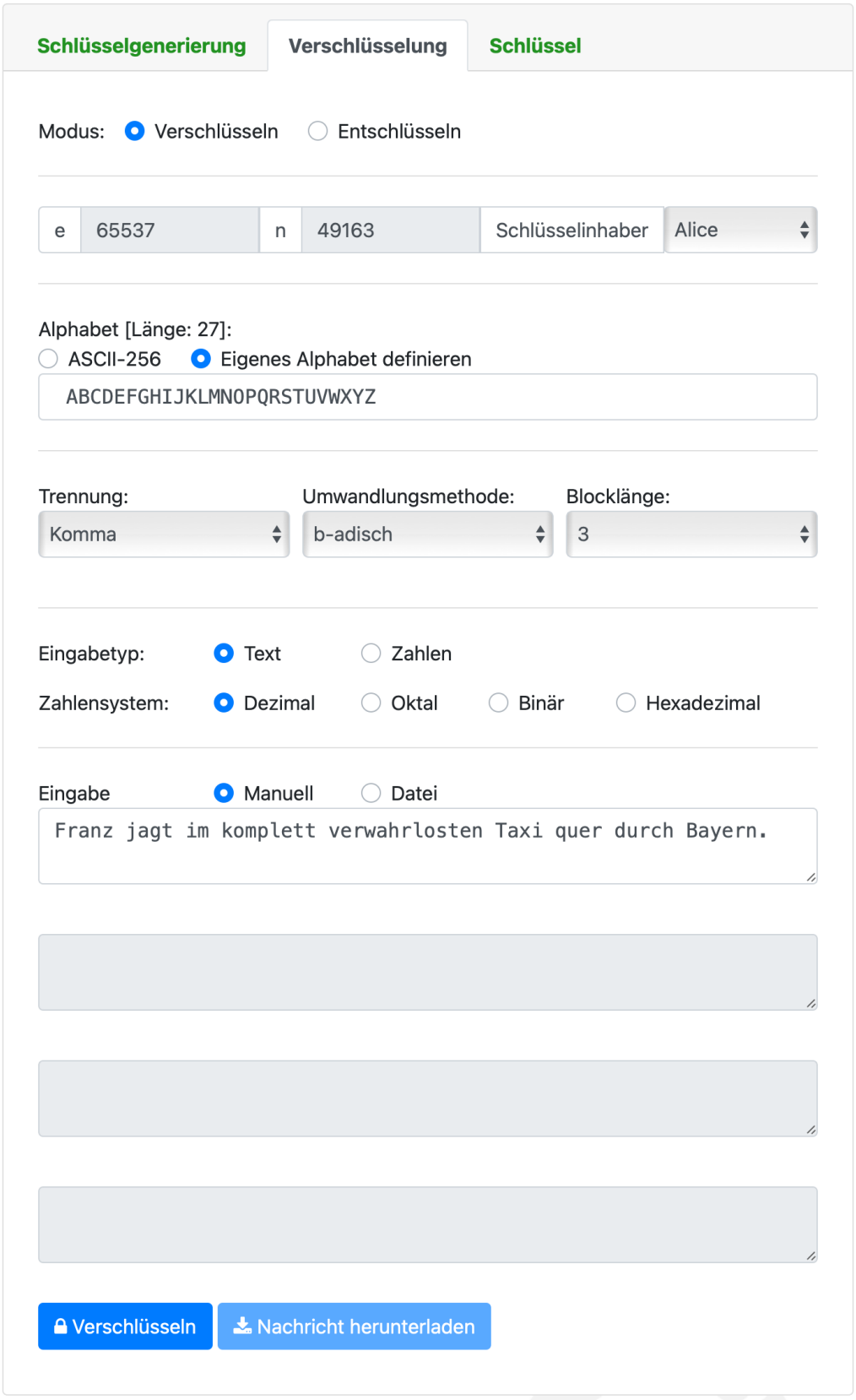

Abbildung A.2: Tab „Verschlüsselung“ in [19]

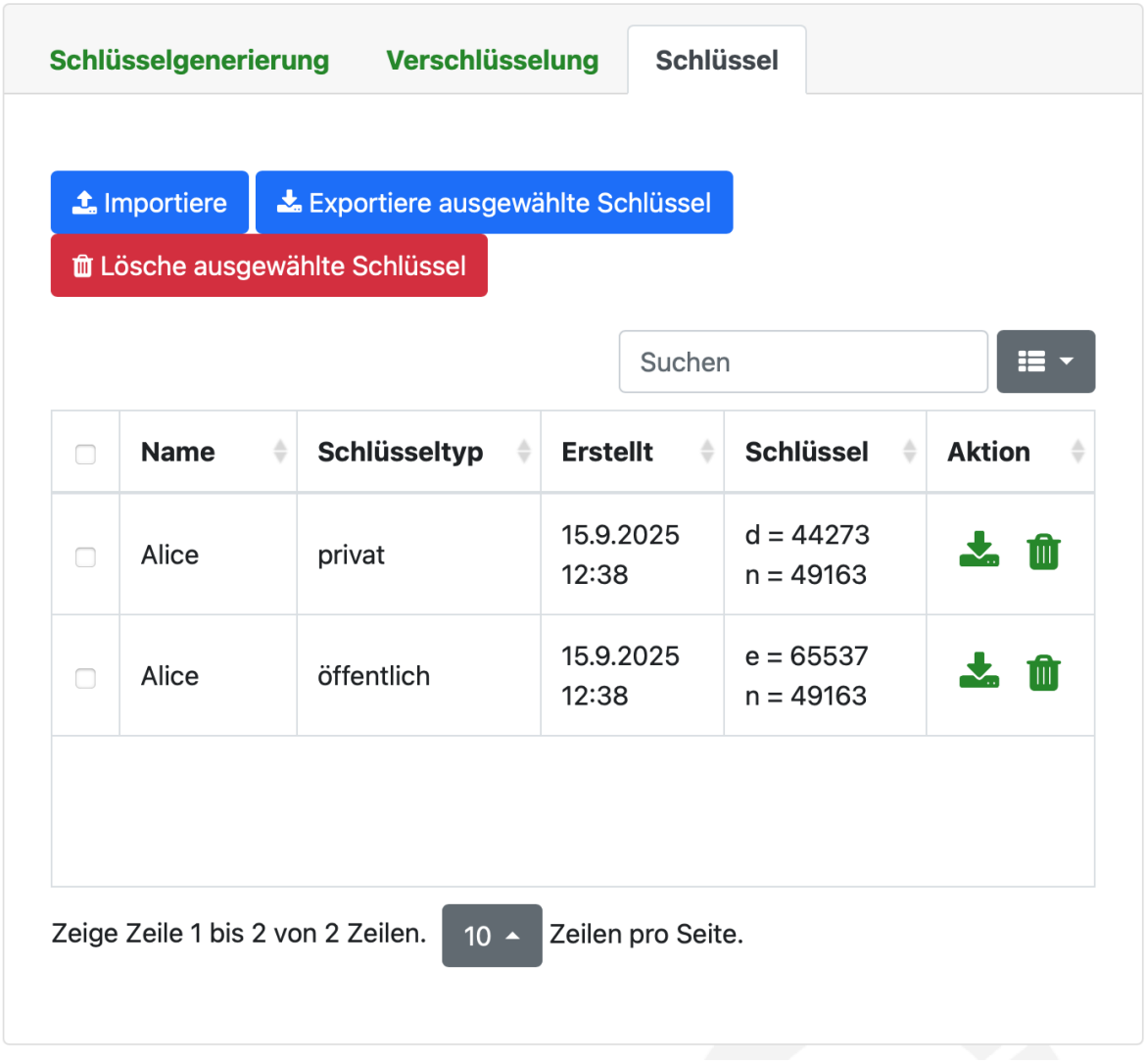

Abbildung A.3: Tab „Schlüssel“ in [19]

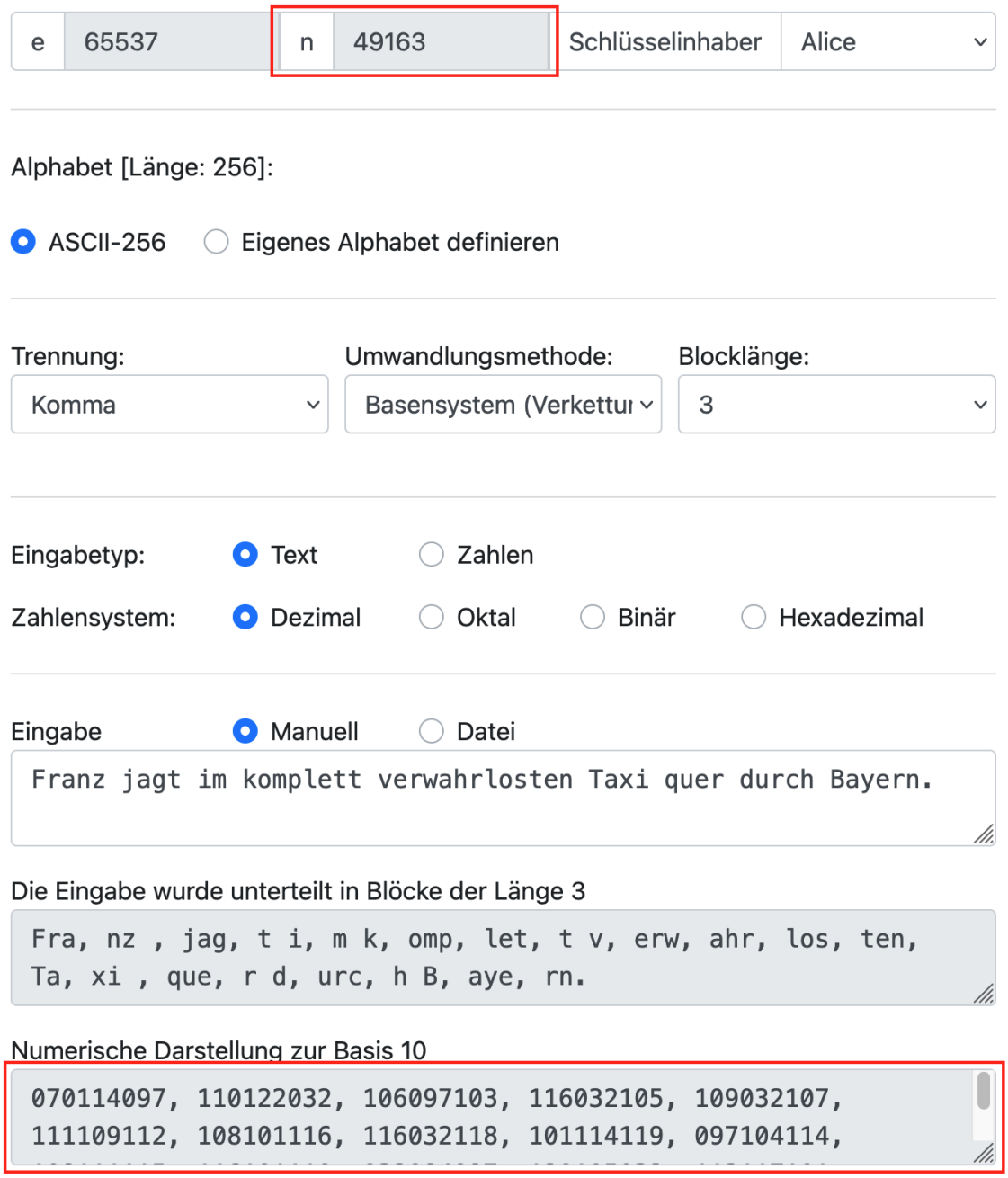

Abbildung A.4: Fehler bei maximaler Blocklänge in [19]

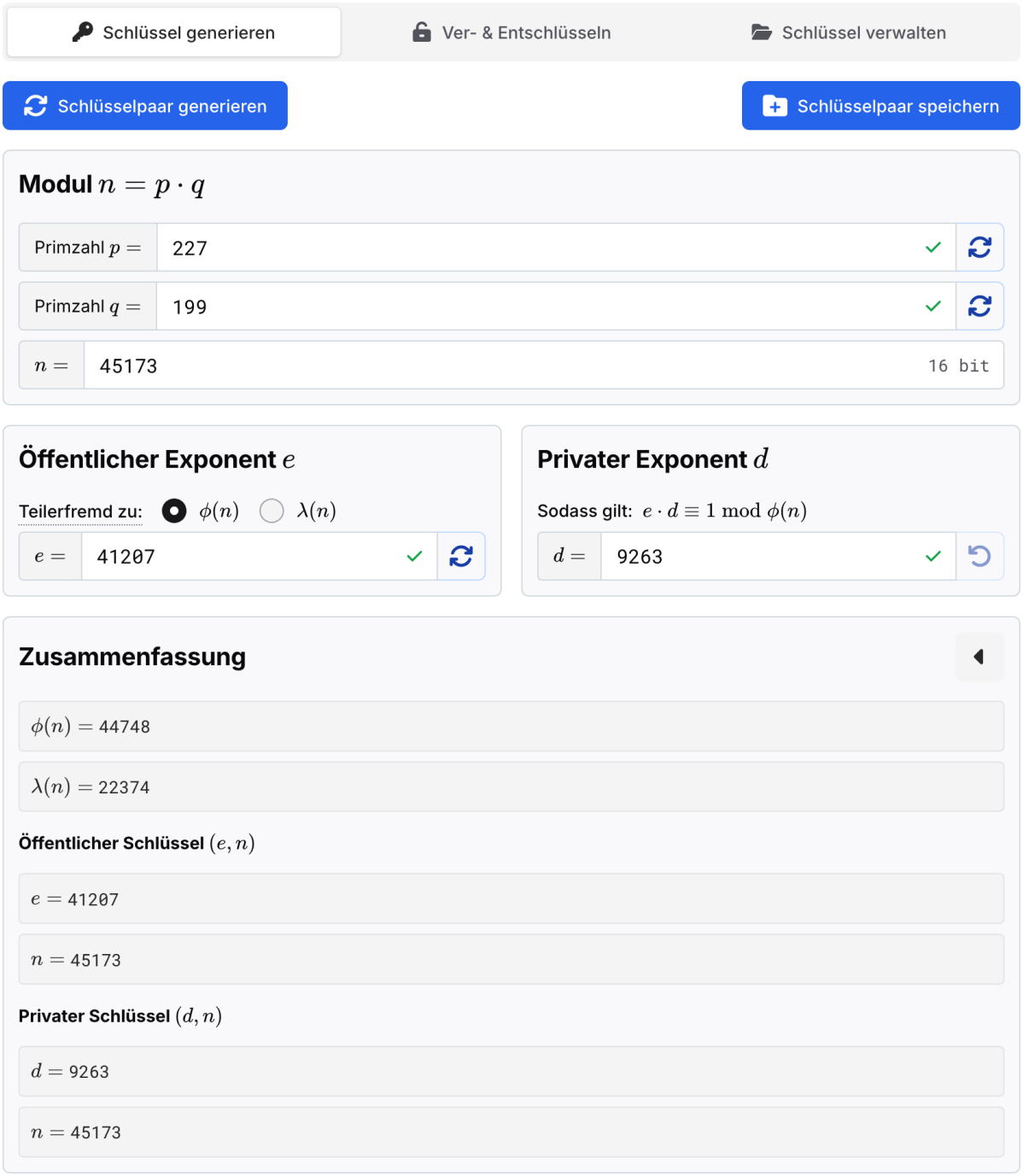

Abbildung A.5: Tab „Schlüssel generieren“ in „RSA didaktisch“

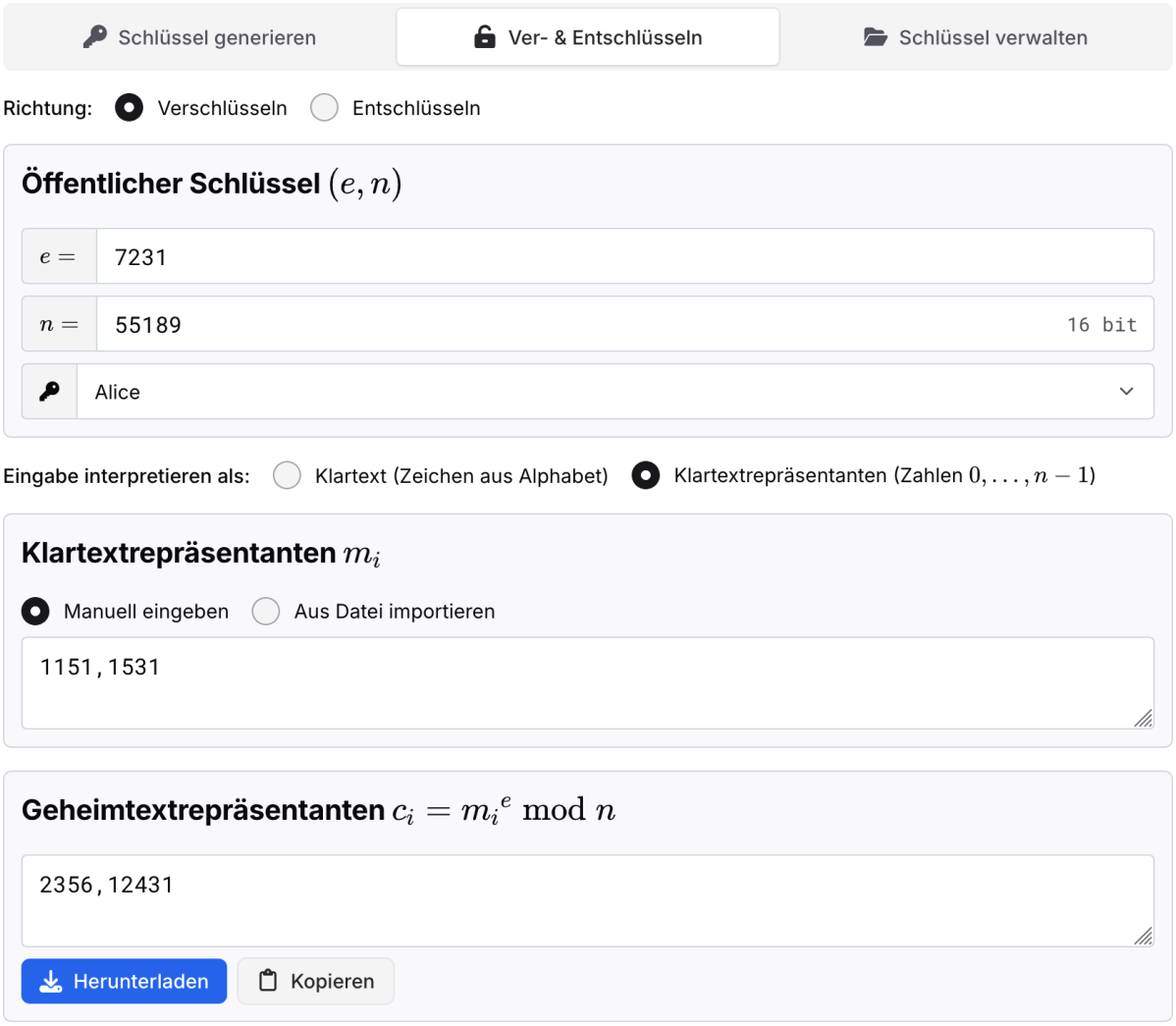

Abbildung A.6: Tab „Ver- & Entschlüsseln“ in „RSA didaktisch“

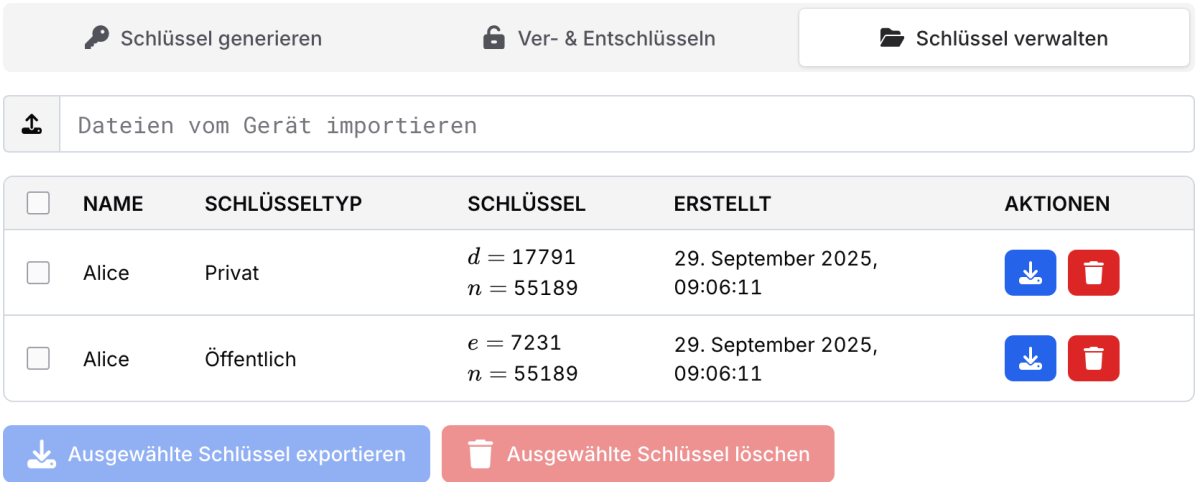

Abbildung A.7: Tab „Schlüssel verwalten“ in „RSA didaktisch“

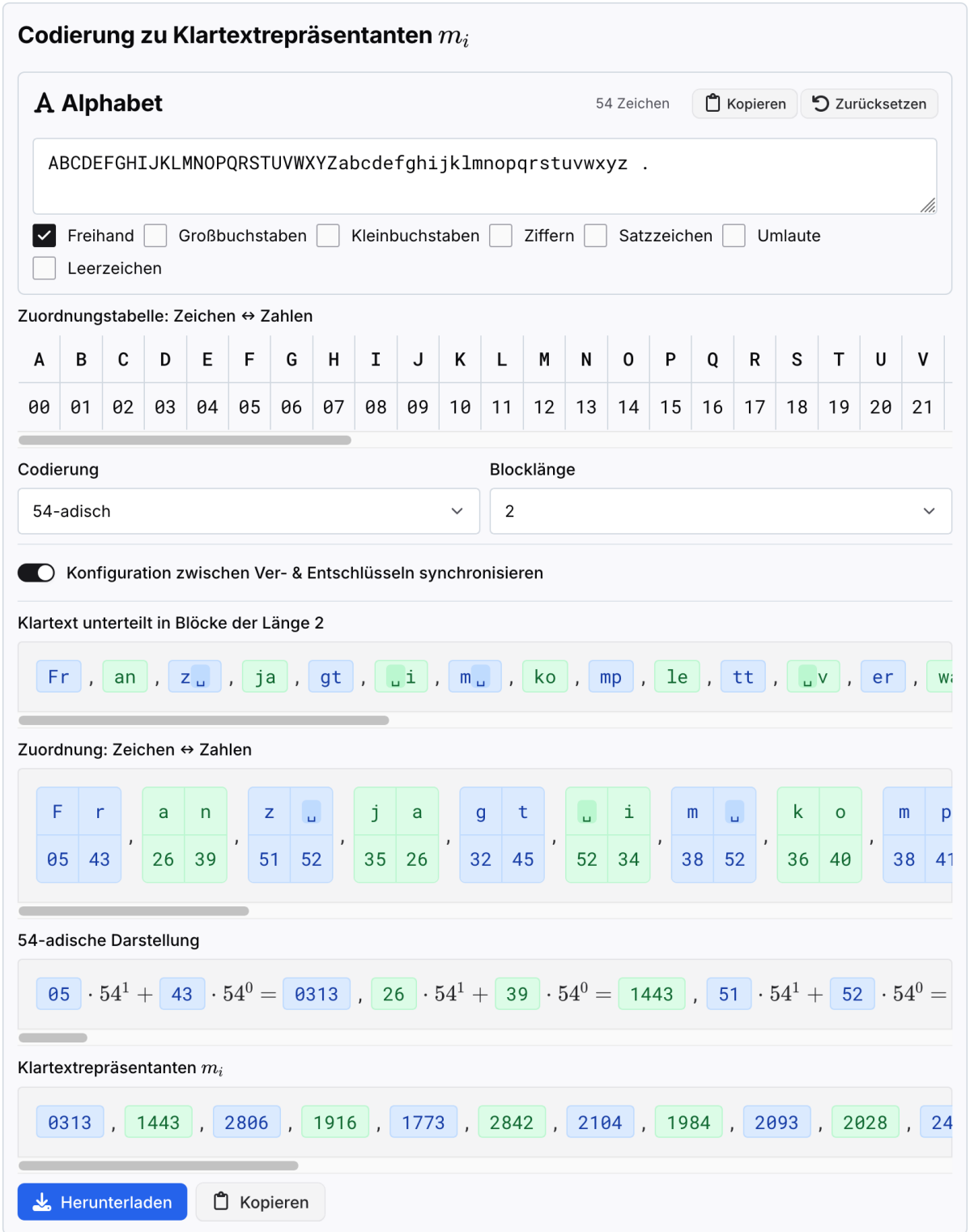

Abbildung A.8: Textcodierung im Tab „Ver- & Entschlüsseln“ in „RSA didaktisch“

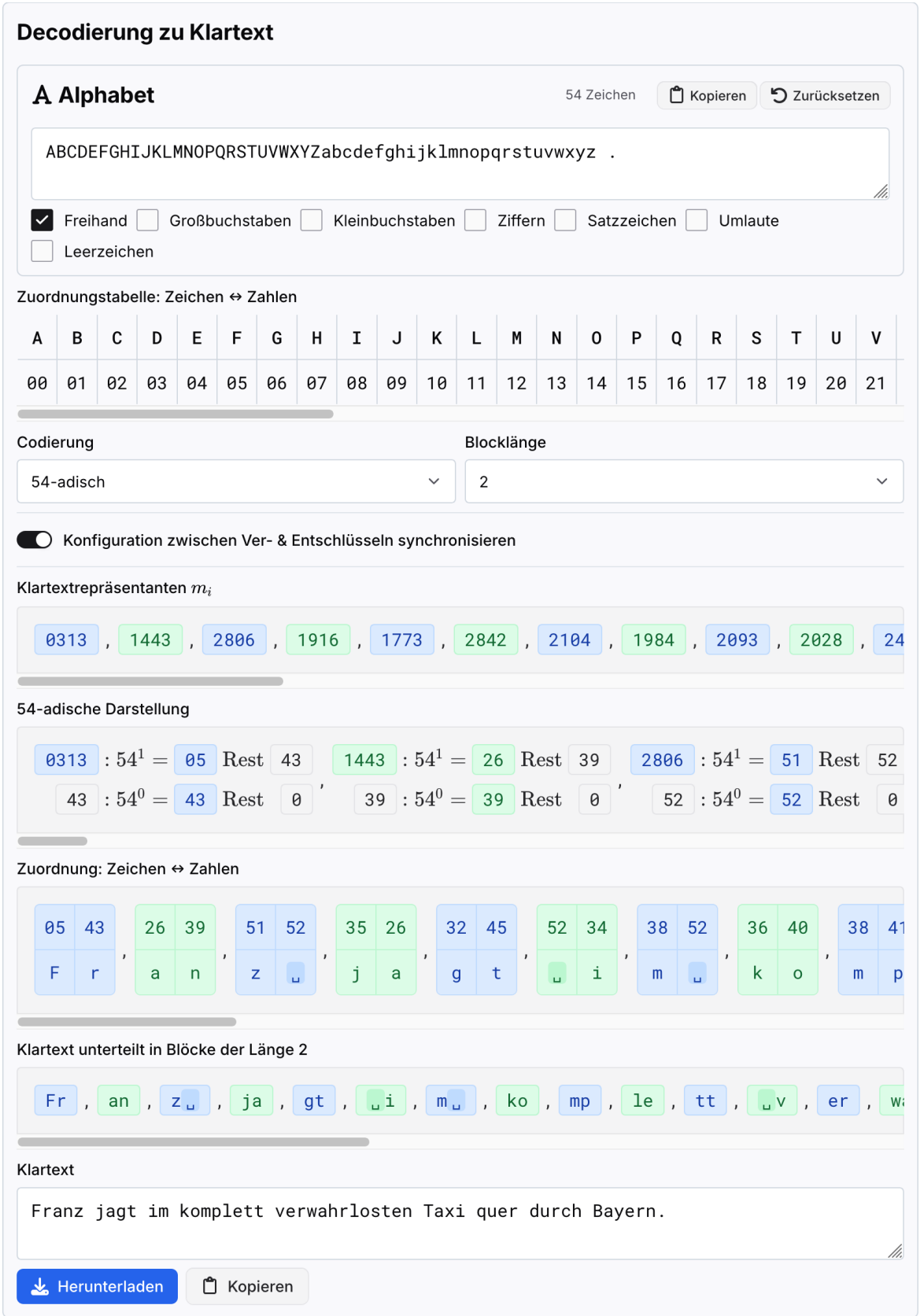

Abbildung A.9: Textdecodierung im Tab „Ver- & Entschlüsseln“ in „RSA didaktisch“

Literaturverzeichnis

[1] Segun Adebayo und Chakra UI Contributors. *Chakra UI*. 2025. URL: https://chakra-ui.com (besucht am 20. 09. 2025) (siehe S. 19, 27).

[2] Divesh Aggarwal und Ueli Maurer. „Breaking RSA Generically Is Equivalent to Factoring“. In: *Advances in Cryptology - EUROCRYPT 2009* (2009). Hrsg. von Antoine Joux, S. 36–53. DOI: 10.1007/978-3-642-01001-9_2 (siehe S. 10).

[3] F. Bahr, M. Böhm, J. Franke und T. Kleinjung. *Factorization of RSA-200*. Lorraine Research Laboratory in Computer Science and its Applications. 2005. URL: https://www.loria.fr/~zimmerma/records/rsa200 (besucht am 20. 08. 2025) (siehe S. 17).

[4] Elaine Barker und Allen Roginsky. *Transitioning the Use of Cryptographic Algorithms and Key Lengths*. Special Publication NIST SP 800-131A Rev. 3. 2024. DOI: 10.6028/NIST.SP.800-131Ar3.ipd (siehe S. 16).

[5] Daniel J. Bernstein, Nadia Heninger, Paul Lou und Luke Valenta. „Post-quantum RSA“. In: *Post-Quantum Cryptography* (2017). Hrsg. von Tanja Lange und Tsuyoshi Takagi, S. 311–329. DOI: 10.1007/978-3-319-59879-6_18 (siehe S. 19).

[6] Dan Boneh, Antoine Joux und Phong Q. Nguyen. „Why Textbook ElGamal and RSA Encryption Are Insecure“. In: *Advances in Cryptology — ASIACRYPT 2000* (2000). Hrsg. von Tatsuaki Okamoto, S. 30–43. DOI: 10.1007/3-540-44448-3_3 (siehe S. 14).

[7] Dan Boneh und Ramarathnam Venkatesan. „Breaking RSA may not be equivalent to factoring“. In: *Advances in Cryptology — EUROCRYPT'98* (1998). Hrsg. von Kaisa Nyberg, S. 59–71. DOI: 10.1007/BFb0054117 (siehe S. 10).

[8] Fabrice Boudot u. a. „Comparing the Difficulty of Factorization and Discrete Logarithm: A 240-Digit Experiment“. In: *Advances in Cryptology – CRYPTO 2020* (2020). Hrsg. von Daniele Micciancio und Thomas Ristenpart, S. 62–91. DOI: 10.1007/978-3-030-56880-1_3 (siehe S. 17, 18).

[9] Daniel R. L. Brown. „Breaking RSA May Be As Difficult As Factoring“. In: *Journal of Cryptology* 29 (2016), S. 220–241. DOI: 10.1007/s00145-014-9192-y (siehe S. 10).

[10] Johannes Buchmann. *Einführung in die Kryptographie*. 6. Aufl. Springer Spektrum Berlin, 2016. ISBN: 9783642397752. DOI: 10.1007/978-3-642-39775-2 (siehe S. 15).

[11] Christopher Chedeau. *React's diff algorithm*. Web Performance Calendar. 2013. URL: https://calendar.perfplanet.com/2013/diff/ (besucht am 30. 12. 2025) (siehe S. 25).

[12] Chakra UI Contributors. *Kbd*. Chakra UI docs. 2025. URL: https://chakra-ui.com/docs/components/kbd (besucht am 30. 12. 2025) (siehe S. 50).

[13] Chakra UI Contributors. *Responsive Design*. Chakra UI docs. 2025. URL: https://chakra-ui.com/docs/styling/responsive-design (besucht am 30. 12. 2025) (siehe S. 28).

[14] Chakra UI Contributors. *Semantic Tokens*. Chakra UI docs. 2025. URL: https://chakra-ui.com/docs/theming/semantic-tokens (besucht am 30. 12. 2025) (siehe S. 28).

[15] Chakra UI Contributors. *Styling*. Chakra UI docs. 2025. URL: https://chakra-ui.com/docs/styling/overview (besucht am 30. 12. 2025) (siehe S. 27).

[16] CrypTool Contributors. *CrypTool-Online*. 2025. URL: https://www.cryptool.org/de/cto/ (besucht am 20. 08. 2025) (siehe S. 5).

[17] CrypTool Contributors. *OpenSSL*. CrypTool-Online. 2025. URL: https://www.cryptool.org/de/cto/openssl/ (besucht am 20. 08. 2025) (siehe S. 5).

[18] CrypTool Contributors. *RSA (Schritt für Schritt erklärt)*. CrypTool-Online. 2025. URL: https://www.cryptool.org/de/cto/rsa-step-by-step/ (besucht am 20. 08. 2025) (siehe S. 5).

[19] CrypTool Contributors. *RSA visuell und mehr*. CrypTool-Online. 2025. URL: https://legacy.cryptool.org/de/cto/rsa-visual (besucht am 20. 08. 2025) (siehe S. 5, 30–35, 59–62).

[20] CrypTool Contributors. *rsa-visual-and-more*. CrypTool-Online Plugin Repository. 2022. URL: https://github.com/cryptool-org/cto/tree/master/_ctoApps/rsa-visual-and-more (besucht am 20. 09. 2025) (siehe S. 43, 51, 52, 55, 56).

[21] Mozilla Corporation. *<length>*. Resources for Developers. 2025. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/length#ch (besucht am 20. 09. 2025) (siehe S. 47).

[22] Mozilla Corporation. *ResizeObserver*. Resources for Developers. 2025. URL: https://developer.mozilla.org/en-US/docs/Web/API/ResizeObserver (besucht am 20. 09. 2025) (siehe S. 48).

[23] Institute of Electrical und Electronics Engineers. *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2019. 2019. DOI: 10.1109/IEEESTD.2019.8766229 (siehe S. 22).

[24] Bernhard Esslinger. *Das CrypTool-Buch: Kryptografie lernen und anwenden mit CrypTool und SageMath*. 2. Aufl. Lehmanns Media, 2025. ISBN: 9783965436114. URL: https://www.cryptool.org/de/ctbook (siehe S. 11, 17).

[25] Niels Ferguson, Bruce Schneier und Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010. ISBN: 9781118722367. DOI: 10.1002/9781118722367 (siehe S. 14).

[26] Benjamin Fine, Anja Moldenhauer, Gerhard Rosenberger, Annika Schürenberg und Leonard Wienke. *Algebra and Number Theory: A Selection of Highlights*. 2. Aufl. De Gruyter, 2023. ISBN: 9783110790283. DOI: 10.1515/9783110790283 (siehe S. 11).

[27] GoogleChromeLabs. *JSBI is a pure-JavaScript implementation of the official ECMAScript BigInt proposal.* 2025. URL: https://github.com/GoogleChromeLabs/jsbi (besucht am 10. 09. 2025) (siehe S. 24).

[28] Daniel M. Gordon. „A Survey of Fast Exponentiation Methods“. In: *Journal of Algorithms* 27.1 (1998), S. 129–146. DOI: https://doi.org/10.1006/jagm.1997.0913 (siehe S. 10).

[29] Web Hypertext Application Technology Working Group. *DOM*. Living Standard. 2025. URL: https://dom.spec.whatwg.org/ (besucht am 30. 12. 2025) (siehe S. 25).

[30] Shu-yu Guo, Michael Ficarra und Kevin Gibbons. *ECMAScript® 2025 Language Specification*. Standard ECMA-262, 16th edition. 2025. URL: https://ecma-international.org/publications-and-standards/standards/ecma-262/ (siehe S. 19–23, 44, 45, 49, 53).

[31] Antoine Joux, David Naccache und Emmanuel Thomé. „When e-th Roots Become Easier Than Factoring“. In: *Advances in Cryptology – ASIACRYPT 2007* (2007). Hrsg. von Kaoru Kurosawa, S. 13–28. DOI: 10.1007/978-3-540-76900-2_2 (siehe S. 10).

[32] Jonathan Katz und Yehuda Lindell. *Introduction to Modern Cryptography*. 2. Aufl. Chapman & Hall/CRC, 2014. ISBN: 9781466570276. DOI: 10.1201/b17668 (siehe S. 10).

[33] Thorsten Kleinjung u. a. „Factorization of a 768-bit RSA modulus“. In: *Advances in Cryptology – CRYPTO 2010* (2010). Hrsg. von Tal Rabin, S. 333–350. DOI: 10.1007/978-3-642-14623-7_18 (siehe S. 17).

[34] Azat Mardan. *React Quickly*. 1. Aufl. Manning Publications Co., 2017. ISBN: 9781617293344 (siehe S. 25, 26).

[35] Meta Platforms, Inc. *JSX*. Specification. 2022. URL: https://facebook.github.io/jsx/ (besucht am 30. 12. 2025) (siehe S. 25).

[36] Meta Platforms, Inc. *React*. The library for web and native user interfaces. 2025. URL: https://react.dev (besucht am 20. 09. 2025) (siehe S. 19, 25, 26).

[37] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson und Andreas Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. Request for Comments 8017. 2016. DOI: 10.17487/RFC8017 (siehe S. 13, 15).

[38] Peter Olson. *Big integer benchmarks*. 2025. URL: https://peterolson.github.io/BigInteger.js/benchmark/#Exponentiation (besucht am 10. 09. 2025) (siehe S. 24).

[39] Peter Olson. *BigInteger.js: An arbitrary length integer library for Javascript*. 2024. URL: https://github.com/peterolson/BigInteger.js (besucht am 10. 09. 2025) (siehe S. 23, 24).

[40] The GnuPG Project. *libgcrypt/cipher/primegen.c*. GNU crypto library. 2025. URL: https://dev.gnupg.org/source/libgcrypt/browse/master/cipher/primegen.c$780 (besucht am 20. 08. 2025) (siehe S. 17).

[41] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Request for Comments 8446. 2018. DOI: 10.17487/RFC8446 (siehe S. 5).

[42] R. L. Rivest, A. Shamir und L. Adleman. „A method for obtaining digital signatures and public-key cryptosystems“. In: *CACM* 21.2 (1978), S. 120–126. DOI: 10.1145/359340.359342 (siehe S. 5, 9).

[43] Florian Rivoal, Håkon Wium Lie, Tantek Çelik, Daniel Glazman und Anne van Kesteren. *Media Queries Level 3*. W3C Recommendation. 2024. URL: https://www.w3.org/TR/mediaqueries-3/ (besucht am 30. 12. 2025) (siehe S. 28).

[44] Christian Schröter, Segun Adebayo und Chakra UI Contributors. *chakra-ui/ark: Build scalable design systems with React, Vue, Solid, and Svelte.* 2025. URL: https://github.com/chakra-ui/ark?tab=readme-ov-file#accessibility-built-in (besucht am 30. 12. 2025) (siehe S. 28).

[45] Peter W. Shor. „Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer“. In: *SIAM Review* 41.2 (1999), S. 303–332. DOI: 10.1137/S0036144598347011 (siehe S. 18).

[46] Bundesamt für Sicherheit in der Informationstechnik. *Kryptographische Verfahren: Empfehlungen und Schlüssellängen*. Technische Richtlinie BSI TR-02102-1. 2025. URL: https://www.bsi.bund.de/dok/TR-02102 (siehe S. 13, 16–18).

[47] Bundesamt für Sicherheit in der Informationstechnik. *Quantum-safe cryptography – fundamentals, current developments and recommendations*. Broschüre. 2021. URL: http://www.bsi.bund.de/dok/pqmigration-en (siehe S. 7, 8, 19).

[48] John A. Smolin, Graeme Smith und Alexander Vargo. „Oversimplifying quantum factoring“. In: *Nature* 499.7457 (2013), S. 163–165. DOI: 10.1038/nature12290 (siehe S. 19).

[49] Douglas R. Stinson und Maura B. Paterson. *Cryptography: Theory and Practice*. 4. Aufl. Chapman & Hall/CRC, 2017. ISBN: 9781315282497. DOI: 10.1201/9781315282497 (siehe S. 7–11, 13, 15, 16).

[50] Robert G. Underwood. *Cryptography for Secure Encryption*. Springer Cham, 2022. ISBN: 9783030979027. DOI: 10.1007/978-3-030-97902-7 (siehe S. 8).

[51] Helmut Witten und Ralph-Hardo Schulz. „RSA & Co. in der Schule. Moderne Kryptologie, alte Mathematik, raffinierte Protokolle. Neue Folge – Teil 3: RSA und die elementare Zahlentheorie“. In: *LOG IN* 28.152 (2008), S. 60–70. URL: https://informatik.schule.de/krypto/RSA_u_Co_NF3.pdf (siehe S. 5).

[52] Paul Wouters, Daniel Huigens, Justus Winter und Niibe Yutaka. *OpenPGP*. Request for Comments 9580. 2024. DOI: 10.17487/RFC9580 (siehe S. 5).

[53] Yaffle. *BigInteger: Yet another implementaion of arbitrary-precision integers in pure JavaScript. Small. Well tested.* 2024. URL: https://github.com/Yaffle/BigInteger (besucht am 10. 09. 2025) (siehe S. 24).

Abkürzungsverzeichnis

ARIA	Accessible Rich Internet Applications
BSI	Bundesamt für Sicherheit in der Informationstechnik
CSS	Cascading Style Sheets
CT	CrypTool
CTO	CrypTool-Online
DOM	Document Object Model
HTML	Hypertext Markup Language
IEEE	Institute of Electrical and Electronics Engineers
JS	JavaScript
JSBI	JavaScript BigInts
JSON	JavaScript Object Notation
JSX	JavaScript Syntax Extension
NIST	National Institute of Standards and Technology
RSA	Rivest-Shamir-Adleman
UI	User Interface
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Tabellenverzeichnis

Codeverzeichnis

Abbildungsverzeichnis

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen aus dem Internet. Diejenigen Paragraphen der für mich geltenden Prüfungsordnungen, die etwaige Betrugsversuche betreffen, habe ich zur Kenntnis genommen.

Der Speicherung meiner Bachelorarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

_____ _____

(Datum) (Unterschrift)

Inhalt der microSD-Karte

- Bachelorarbeit als PDF-Datei
- Quellcode der Web-Anwendung