UNIVERSITÄT SIEGEN

Naturwissenschaftlich Technische Fakultät

Masterarbeit

zur Erlangung des Grades Master of Science Informatik

Implementierung einer interaktiven Benutzeroberfläche für die Kryptoanalyse homophoner Chiffren in CrypTool-Online

Betreuer	Dr. Nils Kopal
Erstprüfer	Prof. Bernhard Esslinger
Zweitprüfer	Dr. Nils Kopal

vorgelegt von	Maik Bastian
Matrikelnummer	902933
Abgabedatum	23. August 2023
Aktualisierung	12. August 2024

Kurzzusammenfassung

Die Ausarbeitung beschreibt die Implementierung einer React-Anwendung zur Kryptoanalyse von homophonen Substitutionschiffren, die in die Webanwendung CrypTool-Online integriert werden soll. Homophone Substitutionschiffren ersetzen jedes Zeichen eines Klartextalphabets durch ein anderes Zeichen aus einer Menge von Zeichen aus einem Geheimtextalphabet. Zur Kryptoanalyse transkribierter Geheimtexte können automatische und manuelle Verfahren eingesetzt werden.

Die implementierte Anwendung bietet die Möglichkeit solche Geheimtexte manuell zu analysieren, indem eine Benutzeroberfläche angeboten wird, die transkribierte Geheimtexte entgegen nimmt und sie mit Hilfe diverser Optionen in die einzelnen Zeichen zerlegt und ansprechend in der Benutzeroberfläche darstellt. Innerhalb dieser Oberfläche können unter anderem die Zeichen des Geheimtextes auf Klartextzeichen abgebildet, vollständig entschlüsselte Wörter markiert und fragliche Entschlüsselungen gekennzeichnet werden.

Nach der Erklärung der Grundlagen und der Konzeption der Anwendung, geht die Ausarbeitung in großem Maße auf die Verarbeitung des Geheimtextes, die Implementierung der Anwendung und die essentiellen Performance-Optimierungen ein. Auf Grund vieler unterstützter Optionen ist die Trennung der Bestandteile der Geheimtexte durch einen Parser umgesetzt, der anhand einer kontextfreien Grammatik die Eingabe überprüft und in einen Ableitungsbaum umwandelt. Dieser Ableitungsbaum wird in der Anwendung mittels diverser Transformationen in die, für die Darstellung, ideale Form gebracht. So müssen neben der Behandlung von Leerraum auch die verschiedenen Gerätetypen, im Sinne eines Responsive-Designs, bedacht werden. Während der Entwicklung sind einige Performance-Probleme aufgetreten, die unter anderem an der feingliedrigen Darstellung der vielen Zeichen von größeren Geheimtexten liegen. Im Zuge der Ausarbeitung werden einige Lösungen für diese Probleme präsentiert.

Abstract

This thesis describes the implementation of a React application for the cryptanalysis of homophonic substitution ciphers, which will be integrated into the CrypTool-Online webapp. Homophonic substitution ciphers replace every symbol of a cleartext alphabet by a symbol out of a set of the ciphertext alphabet. There are automatic or manual methods for the cryptanalysis of transcribed ciphertexts.

The implemented application allows the manual analysis of such ciphertexts, by providing a user interface, that takes transcribed ciphertexts and splits them, under consideration of multiple options, into their individual symbols and presents them in an appealing way. Inside the user interface the symbols of the ciphertext can be mapped to plaintext symbols, completely decrypted words can be marked and doubtful decryptions can be labeled.

After an explanation of the theoretical foundations and the conception of the application, the thesis mostly considers the processing of the ciphertext, the implementation of the application and the essential performance optimizations. Because of the many possible options for splitting the ciphertext into its parts, the processing of the ciphertext is done by a parser, which uses a context-free grammar to evaluate the input and to create a derived tree. To get the derived tree into the expected shape, many transformations are performed on it. These transformations are important for handling whitespaces and different device types with their significance for a responsive design. During development some performance issues arose because of the detailed representation of the many symbols of a large ciphertext. In the thesis some solutions for these issues are presented.

Inhaltsverzeichnis

1 Einleitung

Das CrypTool (CT)-Projekt stellt verschiedenste Applikationen zur Verfügung, mit denen historische und moderne kryptografische Anwendungen eingesetzt und untersucht werden können. Es erleichtert somit die Auseinandersetzung mit Kryptografie und Kryptoanalyse. Aus diesem Grund werden die Anwendungen weltweit eingesetzt. [9]

Im Zuge dieser Ausarbeitung wurde eine Anwendung zur Kryptoanalyse homophoner Substitutionschiffren innerhalb von CrypTool-Online (CTO) entwickelt. Homophone Substitutionschiffren sind klassische Verschlüsselungsverfahren, die die einzelnen Zeichen im Klartext durch ein anderes Zeichen aus einer Menge von Zeichen aus einem Geheimtextalphabet ersetzen. Dazu wird im Schlüssel eine feste Abbildung von je einem Klartextzeichen auf mehrere mögliche Homophone festgelegt, aus denen eine Auswahl bei der Verschlüsselung getroffen werden kann. Die Kryptoanalyse historischer Chiffren beschäftigt sich im Allgemeinen mit der Untersuchung von Geheimtexten. Im Besonderen bei der homophonen Substitution mit dem Analysieren von Mustern im Geheimtext, die die Eigenschaften der Nachricht offenlegen. [1]

1.1 Motivation

Neben den Anwendungen des CT-Projekts für Desktop-Betriebssysteme, gibt es die Webseite CTO, die viele kryptologische Anwendungen bereitstellt. Webanwendungen bieten unter anderem die Vorteile, dass sie ohne Installation einfach und schnell zu nutzen sind, automatische Softwareaktualisierungen erfahren können und keine speziellen Hardware-Anforderungen besitzen [23]. Außerdem finden viele Nutzer den Einstieg in das CT-Projekt über CTO [9]. Ein großes Angebot an Werkzeugen zur Erforschung von kryptografischen und kryptoanalytischen Anwendungen in CTO kann damit das Interesse an dem Projekt begünstigen. Das CTO-Projekt hat sich im Laufe der Jahre immer wieder gewandelt. Die aktuelle Version ist als statische Webseite mit dem *Static-Site*-Generator *Jekyll* umgesetzt, dessen Benutzeroberfläche mit *Bootstrap* erstellt ist und die Implementierung der verschiedenen Algorithmen als Plugins per JavaScript (JS) und *jQuery* erlaubt [5]. Für die nächste Version ist eine Entwicklung mittels *React* und *Next.js* angestrebt. Auch innerhalb der neuen Version von CTO können die einzelnen Anwendungen unabhängig entwickelt werden. Sie bieten aktuell unter anderem Funktionalität zum Ver- und Entschlüsseln mit klassischen und modernen Algorithmen, für Codierungen und zur Kryptoanalyse. React wird in vielen weit verbreiteten Webseiten eingesetzt und erlaubt die Entwicklung von großen, verständlichen und gut wartbaren Applikationen, die anhand eines großen Ökosystems an Bibliotheken und Werkzeugen erweitert werden können [12].

Derzeit sind in der neuen CTO-Seite bereits einige Algorithmen zur Ver- und Entschlüsselung und Codierung implementiert. Der Umfang der Desktop-Anwendungen CrypTool (CT) und CrypTool 2 (CT2) ist deutlich umfangreicher. Ziel ist es den Funktionsumfang des CTO-Projekts zu erweitern. Daher soll im Zuge dieser Ausarbeitung der vorhandene, sogenannte *homophone Substitutionsanalysator* aus CT2 als Webanwendung neu entwickelt werden.

1.2 Ziele

Die zu entwickelnde Anwendung soll eine manuelle Analyse von homophonen Substitutionschiffren mittels einer interaktiven Benutzeroberfläche ermöglichen. Sie soll einfach zu benutzen und gleichzeitig umfangreich genug sein um die hobbymäßige und professionelle Kryptoanalyse zu unterstützen. Dazu soll sie zugänglich sein, ein modernes Design, aber auch weiterreichende Funktionen bieten. Ebenso soll eine Bedienung sowohl auf Desktop-Systemen als auch auf mobilen Geräten möglich sein. Die Anwendung soll die Eingabe von verschieden formatierten Geheimtexten erlauben und diese, anhand von typischen Regeln, in ihre einzelnen Symbole zerlegen. Die Benutzeroberfläche soll die Entschlüsselung der Symbole in Klartext ermöglichen, verschiedene Markierungen der Texte, zur besseren Analyse, erlauben und letztlich eine Ausgabe des Klartextes und Schlüssels erzeugen.

1.3 Aufbau der Arbeit

Zunächst wird in den Grundlagen (Kapitel 2) ein Überblick über monoalphabetische und homophone Substitutionschiffren sowie eine Erklärung der Ansätze der Kryptoanalyse dieser Chiffren gegeben. Außerdem wird ein Überblick über Transkriptionen historischer Geheimtexte gegeben. Diese Grundlagen erläutern die Form der unterstützten Eingaben in der Anwendung. Daneben dient das Kapitel dazu, grundlegende Aspekte der Textanalyse sowie Besonderheiten der Unicode-Unterstützung von Texten aufzuzeigen. Zudem sollen einige Besonderheiten von JS angeführt werden, die vor allem bei der Implementierung der Textanalyse in der Anwendung von Relevanz sind. Zuletzt erfolgt eine Einführung in React, die den grundlegenden Aufbau von Webanwendungen mit dieser Technologie erläutert.

Im nächsten Kapitel (Kapitel 3) sollen die Anforderungen an die entwickelte Anwendung dargestellt werden. Dabei ist der Aufbau der unterstützten Geheimtexte sowie die Gestaltung der Nutzeroberfläche der Schwerpunkt dieses Kapitels.

Die Implementierung (Kapitel 4) beginnt mit einer Beschreibung der fertig gestellten Nutzeroberfläche. Anschließend werden Details der Implementierung aufgeführt. Nach

einem Überblick über die Strukturierung der Anwendung und deren Datenverwaltung, sollen die Verarbeitung des Geheimtextes und die damit verbundene Textanalyse sowie die Unterstützung eines Responsive-Designs in der Anwendung im Detail besprochen werden. Darauf erfolgt ein kurzer Einblick in Ansätze zur Umsetzung einer automatischen Analyse von homophonen Geheimtexten mittels der bestehenden Implementierung aus CT2. Abschließend wird eine Analyse der möglichen Performance-Optimierungen, die eine flüssige Nutzung der Anwendung erst ermöglichen, vorgestellt.

Die Evaluation (Kapitel 5) untersucht die Bedienbarkeit der Anwendung anhand einer Entschlüsselung eines unbekannten Geheimtextes und kleiner Nutzerstudien. Es sollen sowohl positive als auch negative Aspekte beschrieben und Ideen für Verbesserungen der Anwendung gegeben werden.

Anschließend dient die Aufführung verwandter Arbeiten in Kapitel 6 dazu, andere Anwendungen zur Analyse von homophonen Substitutionschiffren zu benennen und sie kurz mit der entwickelten Anwendung zu vergleichen.

Zuletzt wird im Fazit (Kapitel 7) bewertet, wie weit die gesetzten Ziele der Ausarbeitung erreicht wurden und welche Erweiterungen der Implementierung in der Zukunft denkbar wären.

2 Grundlagen

Dieses Kapitel beginnt mit einer Beschreibung der in der Anwendung unterstützten Chiffren und historischen Geheimtexte. Neben einem kurzen allgemeinen Überblick über verschiedene Chiffren, sollen vor allem monoalphabetische und homophone Substitutionschiffren besprochen werden, da diese von der Anwendung verarbeitet werden können. Daraufhin folgt ein Überblick über die Kryptoanalyse, wieder mit einem Fokus auf monoalphabetische und homophone Substitutionschiffren. Die anschließende Beschreibung der Transkription von Dokumenten dient, zusammen mit einer Einführung in den Unicode-Standard, dem Einblick in die zu erwarteten Eingaben der Applikation. Eine Einführung in kontextfreie Grammatiken und der Text-Analyse soll dem besseren Verständnis der späteren Kapiteln helfen. Beide Theorien sind bei der Verarbeitung des Geheimtextes in der Anwendung von Relevanz. Zuletzt sollen mittels eines Überblicks über JS und React die verwendeten Web-Technologien zur Implementierung der Anwendung vorgestellt werden.

2.1 Monoalphabetische und homophone Substitutionschiffren

Bei Substitutionschiffren handelt es sich um Algorithmen, die zum Verschlüsseln ein oder mehrere Zeichen des Klartextes durch ein oder mehrere Zeichen eines Alphabets ersetzen. Wie diese Abbildung genau funktioniert, hängt von der Art der Chiffre ab. Es gibt unter anderem mono- und polyalphabetische Substitutionschiffren. Monoalphabetische Chiffren tauschen Zeichen aus dem Klartextalphabet immer mittels eines Geheimtextalphabets. Bei den polyalphabetischen Chiffren werden mehrere Geheimtextalphabete genutzt, zwischen denen während der Chiffrierung gewechselt wird. [13]

Die Caesar-Chiffre, die zuerst von Julius Caesar beschrieben wurde, ersetzt jedes Zeichen aus dem Klartextalphabet durch ein anderes Zeichen aus demselben Alphabet, dessen Auswahl durch Verschieben der Position des Klartextzeichen um eine feste Distanz getroffen wird. Sie ist somit eine monoalphabetische Substitutionschiffre. Die Distanz vom Klartext- zum Geheimtextzeichen wird durch einen Schlüssel definiert. Hat der Schlüssel den Wert 3, so wird der erste Buchstabe im Alphabet durch den Vierten, der zweite Buchstabe durch den Fünften und so weiter ersetzt. Tabelle 1 zeigt eine solche Abbildung mit Schlüssellänge 3. [7]

Klartextzeichen	A	B	C	D	...	X	Y	Z
Geheimtextzeichen	D	E	F	G	...	A	B	C

Tabelle 1: Abbildung der Caesar-Chiffre mit Schlüssellänge 3

Die *Atbash*-Chiffre ist ein weiteres Beispiel einer monoalphabetischen Substitutionschiffre. Sie kommt aus dem Hebräischen und wurde zum Verschlüsseln biblischer Texte genutzt. Die Verschlüsselung erfolgt durch ein Vertauschen der Ordnung der Buchstaben. Der erste Buchstabe im Alphabet wird auf den letzten Buchstaben abgebildet, der zweite Buchstabe auf den Vorletzten, etc. Die Tabelle 2 verdeutlicht dieses Verfahren. [7]

Klartextzeichen	A	B	C	D	...	X	Y	Z
Geheimtextzeichen	Z	Y	X	W	...	C	B	A

Tabelle 2: Abbildung der Atbash-Chiffre

Im Kapitel 2.2 sollen Möglichkeiten der Kryptoanalyse von homophonen Substitutionschiffren detaillierter beschrieben werden. Hier erfolgt bereits eine kurze Einordnung der Sicherheit von Chiffren anhand der Häufigkeitsanalyse. Beide bisher diskutierten Chiffren sind anfällig gegenüber einer solchen Analyse der Häufigkeit von Buchstaben und Wörtern in einem verschlüsselten Dokument. Die Abbildungen von Klartext-Buchstaben der in einer Sprache am häufigsten vorkommenden Buchstaben, beziehungsweise Buchstabenkombinationen, werden mit hoher Wahrscheinlichkeit in einem, mit diesen Algorithmen verschlüsselten Dokument, ebenfalls am häufigsten vorkommen. So sind im Englischen die Wörter *I* und *a* oder *the* und *and* markant beziehungsweise häufig. [7]

Bei der homophonen Substitution kann ein Klartextzeichen durch ein Zeichen aus einer Menge verschiedener Geheimtextzeichen, den Homophonen, ersetzt werden. Homophon bedeutet „gleich klingend“, da jedes Homophon, welches zu einem Klartextzeichen gehört, als dessen Klang verstanden werden kann. Neben den Zeichen des Klartextalphabets oder mehrerer verbundener Ziffern, können zusätzlich auch eigene Zeichen erfunden werden. So wurden historisch auch andere Zeichenarten, wie zum Beispiel astronomische oder alchemistische Symbole, verwendet [16]. Für eine beispielhafte Abbildung von Klartextzeichen auf Homophone siehe Tabelle 3. Zur Verschlüsselung wird für jedes Klartextzeichen zufällig eines der im Schlüssel vorgesehenen Homophone gewählt. Die Entschlüsselung ist eindeutig, da jedes Homophon nur einmal im Schlüssel vergeben werden darf. Homophone Substitutionschiffren sollen die Häufigkeitsanalyse erschweren. Häufig vorkommende Zeichen müssen dafür im Klartextalphabet durch mehr Homophone ersetzt werden als seltener vorkommende Zeichen. Wie später in diesem Kapitel beschrieben, ist dies nicht immer perfekt umgesetzt. Zunächst gab es in historischen Chiffren lediglich eine kleine Auswahl an Homophonen für einzelne Buchstaben [13]. Für eine ideale Auswahl der Homophone ist die Häufigkeitsanalyse einzelner Buchstaben hilfreich. Im Englischen kommt der Buchstabe *B* zu 2% vor, der Buchstabe *D* mit 3%. Ein Schlüssel einer homophonen Substitutionschiffre könnte so für *D* 1,5-mal so viele Homophone definieren wie für *B*. Während eine Häufigkeitsanalyse der einzelnen Buchstaben mit der homophonen Substitution schwierig ist, ist diese Art von Chiffre anfällig gegenüber Analysen von Buchstabenpaaren. [19]

Klartextzeichen	A	B	C	D	E	...	Y	Z
Geheimtextzeichen	09 81	08	43	11	12 01 67 03 04	... ...	02	39

Tabelle 3: Beispiel zur Abbildung von Klartextzeichen auf Homophone

Erweiterungen von Substitutionschiffren sind sogenannte Nullen (engl. *nulls*) und Nomenklatoren (engl. *nomenclator*). Nullen sind Zeichen im Geheimtextalphabet, die keine Bedeutung besitzen. Sie werden in den Geheimtext eingefügt um Verwirrung bei der unbefugten Entschlüsselung hervorzurufen. Ein Nomenklator ist ein System, bei dem die verwendete Substitutionschiffre um Codes erweitert wird. Diese Codes bilden Wörter, Namen oder Silben auf bestimmte Symbole im Geheimtext ab. Zur erfolgreichen Entschlüsselung wird, neben dem Schlüssel der Substitutionschiffre, ebenfalls die Liste der Codes benötigt. [13]

In der westlichen Welt kommt die erste bekannte Verwendung einer homophonen Substitutionschiffre in der Kommunikation zwischen Simeone de Crema und dem Herzogtum von Mantua im Jahr 1401 vor. Abbildung 1 aus [13] zeigt den für diese Chiffrierung verwendeten Schlüssel. Es ist zu erkennen, dass lediglich für die Buchstaben *a*, *e*, *o* und *u* Homophone vergeben wurden. Erst viele Jahre später kamen deutlich mehr Homophone in den damaligen Chiffren zum Einsatz. Hingegen wurden bereits am Ende des 14. Jahrhunderts die ersten Nomenklatoren eingeführt. Zunächst verwendeten sie monoalphabetische Substitutionen im Zusammenhang mit den Codes, später dann auch homophone Substitutionen. Zunächst gab es lediglich einige dutzend Codes als Verschlüsselung für ganze Wörter. Zu ihrem Höhepunkt bestanden Nomenklatoren aus tausenden Codes für Wörter und einzelne Silben. Nach ihrer Einführung waren Nomenklatoren für rund 450 Jahre zur Sicherung der Kommunikation in Europa und Amerika geläufig. Beispielsweise ließ sich König Philip II von Spanien 1556 eine neue Nomenklator-Chiffre erstellen. Diese gehörte zu den besten Systemen der damaligen Zeit und bildete eine Grundlage für Nomenklatoren in Spanien bis ins 17. Jahrhundert. [13]

Ein Beispiel für polyalphabetische Substitutionschiffren ist die Vigenère-Chiffre [7]. Da diese Art von Substitutionschiffren nicht für diese Ausarbeitung relevant sind, werden sie nicht weiter beschrieben.

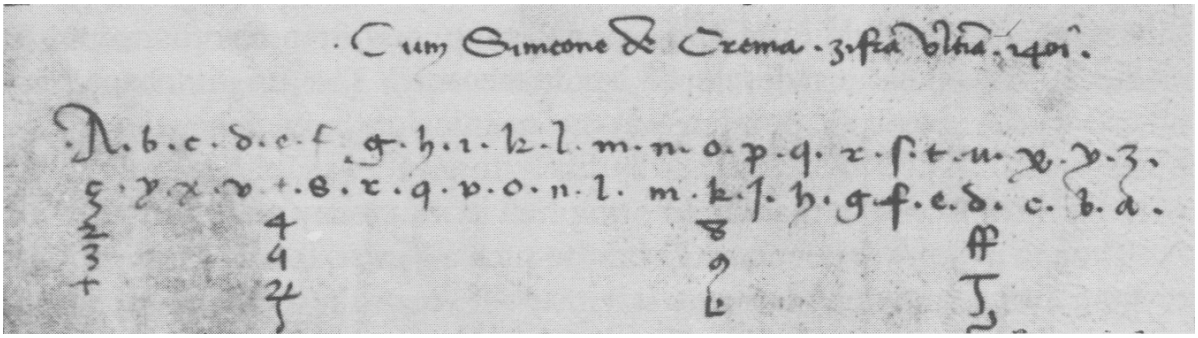

Abbildung 1: Schlüssel der Chiffre zwischen Herzogtum Mantua und Simeone de Crema aus [13]

2.2 Kryptoanalyse von homophonen Substitutionschiffren

Die Kryptoanalyse versucht einen Schlüssel zu einem gegebenen Geheimtext zu finden. Der Versuch soll dabei weniger Aufwand benötigen als ein einfaches Brechen des Kryptosystems durch Ausprobieren aller möglichen Kombinationen. Ein Wissen über die Kryptoanalyse ist für Geheimdienste oder militärisches Personal von Belang, aber auch zur Verifikation eines kryptografischen Algorithmus. [7]

Die nächsten Abschnitte geben einen Überblick über klassische Methoden der Kryptoanalyse, welche unter anderem bei der Untersuchung von monoalphabetischen und homophonen Substitutionschiffren von Bedeutung sind. Außerdem soll zuletzt, anhand eines automatischen Analysealgorithmus, wie er in CT2 implementiert ist, ein moderner Ansatz skizziert werden.

Ein Ausprobieren aller Möglichkeiten eines Schlüssels zu einem Kryptosystem ist nur in trivialen Fällen sinnvoll machbar. Es wird dabei jede Variante eines Schlüssels ausprobiert und der Geheimtext transformiert. Enthält einer der generierten Texte sinnvollen Klartext, ist ein Schlüssel mit hoher Wahrscheinlichkeit gefunden. Es kann aber mehrere mögliche Lösungen für einen Geheimtext geben. Die Unizitätslänge gibt an, ab welcher Länge ein Klartext wahrscheinlich als korrekt erkannt werden kann. Bei komplexeren Chiffren verhindert meist der Rechenaufwand einen solchen *Brute-Force*-Angriff. Die Analyse von monoalphabetischen und homophonen Substitutionschiffren stützt sich auf Eigenschaften der Sprache der originalen Klartextnachricht. Diese Eigenschaften können anhand einer Mustererkennung und Häufigkeitsanalyse untersucht werden. [1]

Bei einfacher monoalphabetischer Substitution bleiben Wiederholungsmuster des Klartextes auch im Geheimtext erhalten. Das Wort „Berggänger“ enthält beispielsweise neben dem dreifach auftretenden Buchstaben „g“ auch die Wiederholung „er“. Wird es durch eine Caesar-Chiffre mit Distanz 3 verschlüsselt enthält „Ehujjaqjhu“ entsprechende Muster. Homophone oder polyalphabetische Substitutionschiffren verringern die Häu-

figkeit solcher Wiederholungen hingegen. Manche Wiederholungsmuster kommen häufiger in Sprachen vor, andere seltener. Nicht immer kann ein solches Muster eine sinnvolle Lösung liefern. Leerraum ist ein weiteres Muster in natürlichen Sprachen. Wird dieser im Geheimtext erhalten, kann die Länge der Wörter entscheidend bei der Analyse helfen. Zwischenraum und Wiederholungen sollten bei der Verschlüsselung ausgelassen werden, um solche Schwachstellen zu vermeiden. [1]

Die Häufigkeit von verschlüsselten Zeichen entspricht bei der monoalphabetischen Substitution der Häufigkeit der entsprechenden Zeichen des Klartextes. Diese Verteilung der Häufigkeit von Buchstaben im Alphabet ist je nach Sprache verschieden. Allerdings gibt es auch innerhalb einer Sprache, je nach analysierter Quelle, Unterschiede in der Häufigkeitsverteilung. Verschiedene Autoren oder verschiedene Themengebiete führen zu unterschiedlicher Häufigkeit der einzelnen Buchstaben. Außerdem muss für eine solche Untersuchung der Geheimtext eine ausreichende Länge besitzen, damit die Analyse sinnvolle Ergebnisse liefern kann. Neben einer Analyse einzelner Zeichen, können auch Multigramme zur Häufigkeitsanalyse verwendet werden. Multigramme sind mehrere Zeichen, die hintereinander vorkommen. Im Deutschen sind die häufigsten Bigramme „ER“, „EN“ und „CH“ und die häufigsten Trigramme „EIN“, „ICH“ und „NDE“. Auch deren verschlüsselte Abbildungen entsprechen wieder der Häufigkeit der entsprechenden Bi- oder Trigramme im Klartext. Durch Multigramme wird eine Sprache noch genauer spezifiziert. Bei dieser Analyse muss ein genügend langer Geheimtext vorhanden sein, um genügend Bi- oder gar Trigramme zur Verfügung zu haben. [1]

Während homophone Substitutionschiffren unanfällig gegenüber einer einfachen Häufigkeitsanalyse sind, können diese Chiffren dennoch mittels Multigramm-Analyse entziffert werden. Buchstabenkombinationen, die mit einem seltenen Buchstaben beginnen, sind relativ einfach erkennbar. So ist im Englischen der Buchstabe Q fast immer gefolgt von dem Buchstaben U. Da Q selten vorkommt, wird er vermutlich nur durch ein Homophon abgebildet. Kommt im Geheimtext ein Homophon vor, auf das immer die gleichen zwei oder drei Homophone folgen, könnte es sich um die Kombination QU handeln. [19]

Als Datenquelle für eine Häufigkeitsanalyse können typische Bücher einer Sprache dienen. Es sollten mehr als 100.000 Buchstaben in einem jeden Werk vorhanden sein, um genaue Werte zu erhalten. Auch eine Auswahl aus vielen Themengebieten ist sinnvoll, da zum Beispiel technische Dokumente eine andere Buchstabenverteilung aufweisen können als literarische Werke. [19]

Der homophone Substitutionsanalysator in CT2 implementiert einen Algorithmus, der mittels *Hillclimbing* und *Simulated Annealing* versucht, automatisch einen Klartext zu dem gegebenen Geheimtext zu finden. Zunächst wird mit einem zufällig generierten Schlüssel begonnen, dessen Einträge entsprechend der Häufigkeit der Buchstaben im Klartextalphabet gewählt werden. In jeder Iteration werden die Abbildungen im Schlüssel mit den Nachbareinträgen vertauscht und dieser so generierte, neue Schlüssel getestet. Gibt es nach 100 Versuchen keine bessere Lösung, werden die letzten drei Abbildungen

des Schlüssels zufällig neu belegt. Der Algorithmus stoppt nach einer definierten Zahl von Versuchen. Die Überprüfung des aktuellen Schlüssels geschieht jeweils durch einen Test, basierend auf der Häufigkeitsverteilung der Pentagramme im mutmaßlichen Klartext. [14]

2.3 Transkription von Geheimtexten

Historische Dokumente, die in Papierform vorliegen, müssen zur digitalen Verarbeitung zunächst in Bildform überführt werden. Anschließend ist eine Umwandlung in reinen Text sinnvoll, um eine Analyse zu ermöglichen. Auch die entwickelte Anwendung erwartet als Eingabe einen Geheimtext in Textform. Bei der Transkription gibt es einigen Spielraum wie unter anderem mit Leerzeichen oder Klartext innerhalb eines verschlüsselten Abschnitts umgegangen wird. Ebenso gibt es unterschiedliche Vorgehensweisen zur Transkription von Zeichen, die nicht zu den üblichen westlichen Zeichensätzen gehören. Es ist daher sinnvoll für die Transkription bestimmte Standards vorzugeben. Das Dokument „Transcription of Historical Ciphers and Keys: Guidelines“[16] des DECRYPT-Projekts[1] definiert solche Richtlinien.

Die DECRYPT-Richtlinien sollen zu großen Teilen in der Anwendung unterstützt werden und sind daher in Kapitel 3.2 genauer vorgestellt. Neben der allgemeinen Vorstellung dient das Kapitel ebenfalls der Darstellung der konzipierten Integration der Richtlinien in die Anwendung.

Abbildung 2 zeigt einen gescannten Geheimtext aus [16], der sowohl römische und griechische sowie alchemistische und astronomische Zeichen verwendet.

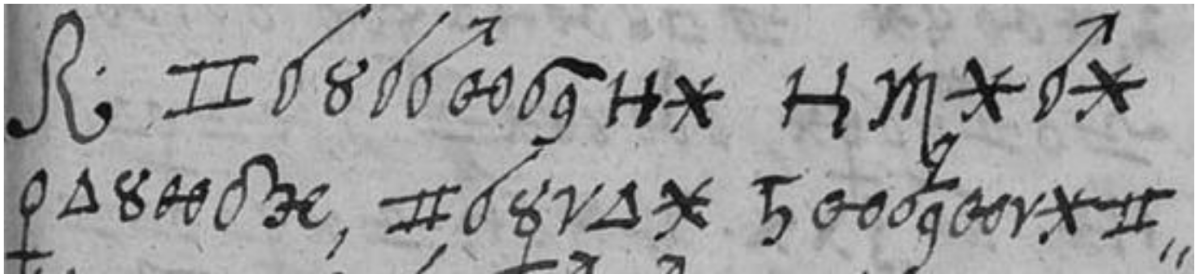

Abbildung 2: Geheimtext mit verschiedensten Symbolen aus [16]

In der folgenden Abbildung 3 ist die Transkription dieses Geheimtextes nach [16] zu sehen. Neben der Umsetzung der Symbole durch visuell darstellbare Unicode-Zeichen sind auch einige Besonderheiten der Richtlinien, wie die Unterstützung von Tags, zu erkennen. Mehr zu diesen Details in Kapitel 3.2.

[1] Für mehr Informationen zum DECRYPT-Projekt siehe: `https://de-crypt.org/`

R <SPACE> ♊♂ ♉♂ ♂ ⊶♋H ∗ <SPACE> H ♍∗ ♂ ∗
♀ △ ♉⊶♁ x? <SPACE> ♊♂ ♉♈△ ∗ <SPACE> 5 ⊶♋⊶♈∗ ♊

Abbildung 3: Transkription der Abbildung 2 mittels Unicode-Zeichen aus [16]

Die erstellte Anwendung orientiert sich an diesen Richtlinien. Unabhängig davon soll die Anwendung genügend Möglichkeiten bieten, um typische Darstellungen von Geheimtexten, die auf homophoner Substitution basieren, zu verstehen. Ein transkribiertes Dokument sollte dabei im Unicode-Format vorliegen, um in der Anwendung analysiert werden zu können.

2.4 Unicode

Unicode ist ein Standard, der die Kodierung von geschriebenen Zeichen und Texten festlegt. Er erlaubt den Umgang mit internationalen Schriftsystemen ohne eine spezielle Behandlung der Zeichen in den verschiedenen Schriftsystemen. Der Standard bietet Unterstützung von mehr als einer Million Zeichen. Neben Zeichen für weltweite Schriften werden auch solche für spezielle Symbole definiert. So gibt es zum Beispiel definierte Zeichen für Währungssymbole, technische Symbole oder *Emojis*.

Die Unterstützung des Unicode-Standards in der entwickelten Anwendung bietet die Handhabung von Texten in verschiedensten Sprachen und Schriftsystemen. Wie in Kapitel 3.2 beschrieben, erlaubt es der Standard ebenfalls die Transkriptionen der Geheimtexte in der Anwendung möglichst detailreich darzustellen. So können bestimmte Markierungen von Zeichen, wie $\dot{3}$, einfach unterstützt werden.

Jedes Zeichen wird durch einen *Code-Point* und einen Namen identifiziert. Die häufigst genutzten Zeichen sind in der *Basic-Multilingual-Plane* enthalten, welche insgesamt 65.536 Code-Points unterstützt. Es werden mehrere Formate definiert, mit denen die Code-Points in eine Sequenz von Symbolen fester Länge umgewandelt werden können. Diese Formate sind UTF-8, UTF-16 und UTF-32. In UTF-8 haben Symbole eine Länge von 8 Bits. In UTF-16 16 Bits und in UTF-32 32 Bits. Alle drei Formate sind miteinander kompatibel, haben für verschiedene Anwendungen aber spezielle Vor- und Nachteile. [4]

Vom Nutzer wahrgenommene Zeichen sind mitunter durch mehrere Code-Points kodiert. Innerhalb des Standards werden diese wahrgenommenen Zeichen als *Grapheme-Cluster* bezeichnet. Das Behandeln dieser Grapheme-Cluster ist wichtig für die korrekte Behandlung von Nutzereingaben, da es anderweitig zu unerwarteten Effekten kommen kann. Diese Effekte können vor allem bei Interaktionen in (graphischen) Benutzeroberflächen, dem Zählen von Zeichen oder der Navigation um einzelne Zeichen auftreten. [6]

Auf Grund all dieser Eigenschaften wird innerhalb der Anforderungsanalyse (Kapitel 3) und der späteren Umsetzung (Kapitel 4.3) auf Unicode-Unterstützung gesetzt.

2.5 Kontextfreie Grammatiken

Zur Verarbeitung des transkribierten Geheimtextes müssen innerhalb der Anwendung bestimmte Regeln vorgegeben sein, welche Zeichenabfolgen in der Eingabe erlaubt sind und wie diese im Verhältnis zu anderen Zeichen stehen. Diese Beschreibung kann mittels kontextfreien Grammatiken (KFG) einfach umgesetzt werden.

Mit Hilfe von KFGs lassen sich unter anderem Analysen von Programmiersprachen innerhalb von Compilern implementieren. In der Umsetzung dieser Ausarbeitung, in Kapitel 4.3, wird des Weiteren die Verarbeitung des Geheimtextes mittels einer KFG implementiert. Grundsätzlich beschreiben Grammatiken, wie die Wörter einer (natürlichen) Sprache korrekt kombiniert werden können. In Programmiersprachen sind die Wörter unter anderem *Identifier*[2], Schlüsselwörter und Operatoren. Diese kleinsten Bestandteile werden im Zusammenhang mit KFGs auch als Terminale oder Token bezeichnet. Zur Überprüfung einer korrekten Eingabe wird zunächst die lexikalische Analyse zur Erkennung dieser Wörter durchgeführt. Die daraufhin ausgeführte syntaktische Analyse bestimmt anschließend, ob der Text die korrekte Grammatik nutzt. [11]

Nach [11, S.76f] besteht eine KFG $G = (N, T, P, S)$ aus einer endlichen Menge von Nonterminalen N, einer endlichen Menge von Terminalen T, wobei $(N \cap T = \emptyset)$ gilt sowie einer endlichen Menge an Produktionen $P \subseteq N \times (N \cup T)^*$ und dem Startsymbol $S \in N$. Produktionen sind also Teilmengen ($\subseteq$) der Mengenprodukte ($\times$) der Mengen der Nonterminalen mit den möglichen Konkatenationen (*) der Vereinigung ($\cup$) der Mengen der Nonterminalen und der Terminalen.

Die Terminale bestimmen das Alphabet der Grammatik und bestehen aus Token, also den erkannten Bestandteilen der Eingabe nach der lexikalischen Analyse. Die Nonterminale stellen die Namen der Regeln der Grammatik dar. Anhand der Produktionen wird beschrieben, wie Nonterminale beim Auffinden von Terminalen in der Eingabe abgeleitet werden sollen. [11]

Eine KFG kann ohne zusätzlichen Kontext erkennen, ob eine Eingabe Teil einer definierten Sprache ist. Hierfür werden die Produktionen angewandt bis die gesamte Eingabe verarbeitet ist. Gibt es keine passende Produktion, entspricht die Eingabe nicht der erwarteten Sprache. Neben diesem reinen Test kann mittels eines Ableitungsbaums auch die Ableitung visualisiert werden. In einem Ableitungsbaum besteht die Wurzel aus dem

[2] zum Beispiel Namen von Variablen oder Konstanten

Startsymbol der Grammatik. Die Kinder der Wurzel sind die Nonterminale oder Terminale der angewandten Produktionen. [11]

Die Produktionen aus [11, S.77] sollen im Folgenden als Beispiel dienen:

```
AExpr  ::= AExpr + Term
        |  Term
Term   ::= Term * Factor
        | Factor
Factor ::= ( AExpr )
        | id
        | num
```

Dieses Beispiel veranschaulicht die Produktionen einer Grammatik für einfache arithmetische Ausdrücke. `AExpr` ist per Konvention das Startsymbol der Grammatik. Mittels `::=` wird eine Produktion definiert. Auf der linken Seite werden die Nonterminale, die mittels der Produktion abgeleitet werden sollen, aufgelistet. Die rechte Seite besteht aus einer Kombination von Nonterminalen und Terminalen. Auch hier wird per Konvention definiert, dass alle Symbole, welche nicht auf der linken Seite der Produktionen vorkommen, Terminale der Grammatik sind. Anhand von `|` werden verschiedene Optionen einer Produktion getrennt. Des Weiteren gibt es für Produktionen die Schreibweise $A \rightarrow \alpha$, wobei α aus $(N \cup T)^*$ besteht. Die erste Produktion $AExpr \rightarrow AExpr + Term \mid Term$ bietet also die Option das Nonterminal $AExpr$ durch $AExpr + Term$ oder $Term$ zu ersetzen. [11]

Obwohl KFGs für die Definition von Programmiersprachen sinnvoll sind, können nicht alle Eigenschaften von Programmiersprachen mit ihnen abgebildet werden. So ist zum Beispiel für den Test, ob Variablen vor ihrer Verwendung definiert wurden, eine spezielle Behandlung im Compiler nötig. [11]

2.6 Text-Analyse

Die folgenden Techniken zur Analyse von Texten werden in der entwickelten Anwendung ausgiebig eingesetzt, um die Eingabe anhand einer definierten KFG zu überprüfen und zu strukturieren. Das Kapitel 4.3 erklärt den Aufbau der KFG und der Implementierung der Text-Analyse im Detail.

Die lexikalische Analyse eines Textes, also die Einteilung in grundlegende Bestandteile, sogenannte Token, wird durch den *Scanner* durchgeführt. In Programmiersprachen müssen zum Beispiel Namen von Variablen, Operatoren, Schlüsselwörter und viele weitere

Bestandteile erkannt werden. Neben der händischen Implementierung eines Scanners, ist eine automatische Erzeugung mittels eines Scannergenerators möglich. Als Eingabe für einen solchen Generator dienen reguläre Ausdrücke, welche die erwünschten Token beschreiben. [11]

Die anschließende syntaktische Analyse wird durch den *Parser* ausgeführt. Diese dient der Überprüfung der korrekten Zusammensetzung der gefundenen Token. Das sogenannte *Parsen* der Eingabe überprüft damit, ob die Tokenfolge der gegebenen Grammatik der Sprache entspricht. Zur Implementierung eines Parsers gibt es unter anderem die Umsetzung als *Recursive-Decent-* oder *Shift-Reduce*-Parser. Ein Recursive-Decent-Parser verfolgt einen Top-Down Ansatz und ist einfach zu implementieren. Jede Produktion eines Nonterminals wird durch eine eigene Prozedur umgesetzt und die gefundenen Terminale in der Eingabe der Reihe nach abgearbeitet. Um die richtige Auswahl der rechten Seite einer Produktion treffen zu können, werden in einem solchen Parser *first-* und *follow*-Mengen berechnet. $first(\alpha)$ gibt das erste Zeichen zurück, mit dem α beginnen kann. Mittels $follow(A)$ wird die Menge der Token, die auf A folgen können ermittelt. Somit kann anhand des aktuellen Nonterminals und des aktuellen und folgenden Tokens aus der Eingabe die Auswahl zwischen B oder C in der Produktion $A \rightarrow B \mid C$ getroffen werden. Probleme bereiten dem Recursive-Decent-Parser allerdings Grammatiken mit Linksrekursion. $A \rightarrow A \mid B$ ist eine solche Produktion, die den Parser zu einer Endlosrekursion verleiten würde. Als bessere, aber auch komplexere Alternative bieten sich Shift-Reduce-Parser an. Da die Implementierung aufwendig ist, gibt es Werkzeuge die die automatische Erstellung eines solchen Parsers übernehmen. [11]

Die Implementierung des Parsers in dieser Ausarbeitung basiert auf den Ideen eines Recursive-Decent-Parser. Die KFG zur Erkennung der Struktur des Geheimtextes besitzt keine Produktionen mit Linksrekursion und kann somit durch diese einfachere Implementierung umgesetzt werden.

2.7 JavaScript

Im Folgenden soll ein sehr kurzer Überblick über JS gegeben und einige besondere Merkmale angesprochen werden, die unter anderem Relevanz für die Verarbeitung des Geheimtextes in Kapitel 4.3 haben.

JS wurde zunächst für die Entwicklung im Web bei *Netscape* konzipiert. Die Sprache wird von allen gängigen, modernen Browsern unterstützt und bietet seit 2010 mittels *Node.js*[3] auch die Möglichkeit Anwendungen direkt auf einem lokalen Computer auszuführen. Es ist eine interpretierte Sprache, die sowohl objektorientierte als auch funktionale Konzepte unterstützt. Die Typisierung erfolgt dynamisch. Der Funktionsumfang der Standardbi-

[3]Node.js Homepage: `https://nodejs.org/en`

bliothek ist sehr beschränkt und wird durch die Laufzeitumgebung im Browser (oder von Node.js) ergänzt. Die Sprache ist bei der ECMA (European Computer Manufacturers Association) unter dem Namen ECMAScript standardisiert. Versionen von JS werden daher als ES5, ES6 und später als ES2016, ES2017 usw. benannt. [10]

Strings werden in JS als UTF-16 kodierte Unicode-Strings aufgefasst. Die 16-Bits reichen zur Darstellung der häufigsten Unicode-Zeichen innerhalb der Basic-Multilingual-Plane. Die meisten Operationen auf Strings nutzen allerdings nur die 16 Bit langen Werte, beachten somit nicht umfangreichere Unicode-Zeichen, wie Grapheme-Cluster. Daher können Operationen auf solchen Zeichen zu unerwarteten Ergebnissen führen. [10]

Das folgende Beispiel in Listing 1 zeigt zunächst, wie das Abfragen der Länge eines Strings mit nur einem sichtbaren Zeichen, eine unerwartete Ausgabe erzeugt. Anstelle der erwarteten Länge von 1, gibt die Eigenschaft `String.length` die Länge 7 für das Emoji 👨‍🎓 aus. Anhand des Aufteilens des Strings mittels der Funktion `split()` kann die interne Darstellung der Zeichen betrachtet werden.

```
1  "👨‍🎓".length // => 7
2  "👨‍🎓".split("") // => ["\ud83e", "\uddd1", "\u83c", "\udffb", "", "\ud83c
     ", "\udf93"]
```

Listing 1: Unicode Grapheme-Cluster in JS

Seit der Version ES6 gibt es mit Generatoren eine Funktionalität, die es erlaubt auf einfache Weise Iteratoren zu erstellen sowie Berechnungen schrittweise durchzuführen. Iteratoren können in JS auf vielfältige Weise genutzt werden. So können die Ergebnisse des Iterators einfach in einer for/of Schleife oder mittels des Spread-Operators[4] verwendet werden. Bei der Nutzung eines Iterators gibt es den Vorteil, dass dieser *lazy* arbeitet, Werte also erst dann liefert, wenn sie vom Aufrufer benötigt werden. Damit kann Speicher gespart und unnötige Berechnungen vermieden werden. Ein Iteratorobjekt ist jedes Objekt, das eine Methode mit dem Namen `next()` anbietet. Diese Methode muss Iterationsergebnisobjekte zurückgeben, welche die beiden Eigenschaften `value` und `done` besitzen sollten. Mittels Generatorfunktionen (`function*`) und des Schlüsselworts `yield` können Funktionen Werte in Form eines Iterators zurückgeben. Eine Generatorfunktion wird beim Aufruf nicht direkt vollständig ausgeführt. Stattdessen pausiert die Ausführung beim Antreffen eines `yield`. Der Wert hinter dem Schlüsselwort wird dann von der Funktion `next()` des Generator-Iterators zurückgegeben. Eine weitere Ausführung der Funktion wartet anschließend wieder auf einen Aufruf der Funktion `next()`. [10]

Im folgenden Listing 2 wird eine Generatorfunktion definiert, die eine absteigende Sequenz von Zahlen, startend bei `from` bis einschließlich `0`, generiert. Beim `yield`-Schlüsselwort wartet die Laufzeitumgebung bis der nächste Wert vom Iterator abgefragt wird.

[4] Mehr Informationen zu dem Spread-Operator unter: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

```
function* countdown(from) {
  for (let i = from; i >= 0; i--) {
    yield i
  }
}
```

Listing 2: Definition einer Generatorfunktion

Wird die Generatorfunktion `countdown`, wie im nächsten Beispiel in Listing 3, mittels for/of Schleife oder mit dem Spread-Operator genutzt, so wird die gesamte Sequenz generiert.

```
const from10 = countdown(10)
for (const number of from10) {
  console.log(number) // prints values 10 to 0
}
const numbers = [...countdown(3)]
console.log(numbers) // prints values: 3, 2, 1, 0
```

Listing 3: Verwendung der Generatorfunktion

Die direkte Nutzung mittels der Funktion `next()` führt dazu, dass nur ein Teil der Werte berechnet wird. Die folgende Generatorfunktion `countdown2` in Listing 4 unterscheidet sich von `countdown` um eine Debug-Logausgabe. Da auf ihrem Iterator nur zweimal die Funktion `next()` aufgerufen wird, wird auch die Schleife nur zweimal durchlaufen.

```
function* countdown2(from) {
  for (let i = from; i >= 0; i--) {
    console.debug(i) // prints values 10 and 9 in debug log
    yield i
  }
}

const iterator = countdown2(10)
const result1 = iterator.next() // result1: { done: false, value: 10 }
const result2 = iterator.next() // result2: { done: false, value: 9 }
```

Listing 4: Generatorfunktion mit Debug-Ausgabe

2.8 Webanwendungen mit React

Die entwickelte Anwendung nutzt *React* ausgiebig um die Benutzeroberfläche zu implementieren. Neben den allgemeinen Konzepten zur Strukturierung der Oberfläche, sind auch einige Besonderheiten des Datenflusses für die Implementierung in Kapitel 4 relevant. Diese Details von React sollen daher im Folgenden behandelt werden.

Die JS-Bibliothek React wurde von Facebook entwickelt und ist flexibel auf verschiedenen Plattformen für die Erstellung von Anwendungen einsetzbar. So können nicht

nur Single Page Application (SPA)-Webanwendungen mit React erstellt, sondern auch vorhandene Webseiten erweitert oder mittels *React Native* Apps für mobile Betriebssysteme wie iOS oder Android entwickelt werden. In Webumgebungen ist es nicht nötig die HTML-Seite ausschließlich auf dem Client aufzubauen. Auch bereits auf einem Server ist es möglich statisches HTML zu erzeugen. Erst bei der Aktualisierung der Anwendung auf dem Client muss dann der JS-Code ausgeführt werden, um die neuesten Änderungen darzustellen. [12]

Komponenten sind grundlegender Bestandteil von Anwendungen in React. Mit ihnen kann ein Ausschnitt einer Benutzeroberfläche beschrieben werden. Dank der JS-Erweiterung JSX sieht diese Beschreibung ähnlich zu HTML aus. JSX-Ausdrücke werden direkt in den JavaScript-Code eingebettet und erlauben es Variablen aus dem umgebenen Code in die Benutzeroberfläche zu integrieren. Durch die Komposition von Komponenten zu komplexen Hierarchien können immer aufwändigere Oberflächen erstellt werden. Die Beschreibung einer Komponente geschieht auf deklarative Art. Es wird also definiert, wie eine Ansicht aussehen soll und nicht welche Schritte ausgeführt werden müssen, um diese zu erstellen. Komponenten verhalten sich dabei funktional. Das heißt, dass die selben Eingabewerte zu derselben grafischen Ausgabe führen. React ist dafür zuständig, die Eingabe und das beschriebene JSX zu verwenden, um das DOM[5] dahingehend zu aktualisieren, dass die HTML-Seite das beschriebene Aussehen wiedergibt. Durch dieses Vorgehen können eine Reihe von Fehlern vermieden werden. Vor allem typische Programmierfehler, die zu inkonsistenten Zuständen führen, sind somit vermeidbar. Dank der funktionalen Natur der Komponenten ergibt sich eine einfachere Testbarkeit, da ausschließlich durch Übergabe der gewünschten Parameter ein Ausschnitt der Benutzeroberfläche generiert und auf das gewünschte Aussehen getestet werden kann. Die funktionale Umsetzung einer Komponente wird dadurch realisiert, dass Änderungen an den Parametern oder dem Zustand einer Komponente zu einem vollständigen Update führen. Neben den übergebenen Parametern besitzen Komponenten auch einen internen Zustand (*State*), der nur innerhalb der Komponente verfügbar ist. Auch eine Änderung dieses Zustands aktualisiert die gesamte Komponente und ihrer Kinder. Änderungen an den Daten führen damit direkt zu einer aktuellen Darstellung der Nutzeroberfläche. Neben der eigentlichen Komponente werden ebenfalls die Kinder der Komponente aktualisiert. Der Datenfluss verläuft somit immer nur in eine Richtung, womit eine weitere Fehlerunanfälligkeit umgesetzt werden soll. [12]

Komponenten können entweder durch Klassen oder Funktionen in JS implementiert werden. Die mittels Funktionen implementierten Komponenten werden auch funktionale Komponenten oder Funktionskomponenten genannt. Seit 2019 (React-Version 16.8) können mittels Funktionen vollwertige Komponenten umgesetzt werden. [12]

Im Folgenden soll der Lebenszyklus einer Komponente anhand eines Beispiels genauer beleuchtet werden. Die Komponente `Counter` in Listing 5 ist durch eine norma-

[5] Document Object Model; siehe: `https://developer.mozilla.org/en-US/docs/Glossary/DOM`

le JS-Funktion umgesetzt. Mittels `export default` wird die Komponente für andere JS-Module sichtbar gemacht.[6] Durch den Parameter `props` können Werte von Eltern-Komponenten an die Komponente übergeben werden. Der Aufruf von `React.useState(0)` gibt zwei Werte zurück. Eine Referenz auf den aktuellen Wert des Zustands (hier `count` genannt) und eine Funktion zum Ändern des Zustands. Nachdem `count` zunächst mit dem initialen Wert `0` initialisiert wurde, wird jede Änderung mittels `setCount()` auch über Funktionsaufrufe hinweg gespeichert. Eine einfache Variable zum Speichern des Zustands ist nicht hinreichend, da – wie oben beschrieben – Änderungen zum erneuten Ausführen des Komponenten-Codes führen. Der aktuelle Wert der Variable würde somit durch die erneute Initialisierung überschrieben. Des Weiteren kann React durch die *set*-Funktion mitgeteilt werden, dass sich der Zustand der Komponente geändert hat. Die beiden Funktionen `incrementCount` und `decrementCount` ändern den internen Zustand der Komponente durch Aufruf von `setCount`. Zuletzt werden die gewünschten HTML-Elemente der Komponente mittels JSX erstellt und von der Funktion zurückgegeben. In dem Beispiel ist das Einfügen von Variablen in JSX (`{count}`) und das Binden von Funktionen an Ereignisse (unter anderem `onClick={incrementCount}`) zu sehen. Mittels `<>` und `</>` wird ein einzelnes Element von der Funktion zurückgegeben ohne den tatsächlichen DOM zu verändern. Andernfalls müsste zum Beispiel ein `div`-Element genutzt werden, um die Beschränkung auf maximal ein Rückgabe-Element innerhalb von Komponenten zu umgehen.[7]

```
1  export default function Counter(props) {
2    const [count, setCount] = React.useState(0)
3
4    function incrementCount(event) {
5      setCount(prev => prev + 1)
6    }
7
8    function decrementCount(event) {
9      setCount(prev => prev - 1)
10   }
11
12   return (
13     <>
14       <p>Current count is: {count}</p>
15       <button onClick={incrementCount}> + </button>
16       <button onClick={decrementCount}> - </button>
17     </>
18   )
19 }
```

Listing 5: Einfache React-Komponente

[6] Für weitere Informationen zu dem Export-Mechanismus in JS siehe: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

[7] Informationen zu der *Fragment*-Komponente von React und den Rückgabe-Beschränkungen in Bezug auf JSX: https://react.dev/reference/react/Fragment

Klickt der Nutzer nun auf den Button +, wird die Funktion `incrementCount` ausgeführt. Diese ruft `setCount` auf, um den Zustand der Komponente zu aktualisieren. Hierdurch wird die Komponente neu berechnet, die Funktion `Counter` also erneut ausgeführt. Anders als beim ersten Darstellen der Komponente ist der Wert von `count` nun `1`. Dieser Wert wird im JSX auch in die HTML-Elemente integriert, so dass im Paragraph der Text „Current count is: 1“ zu sehen ist.

`useState` ist ein Beispiel für eine React-*Hook*. Hooks integrieren Funktionalität von React in Funktionskomponenten und beachten dabei den Lebenszyklus dieser Komponenten. Per Konvention starten alle Hooks mit dem Präfix `use`. React bietet neben `useState` unter anderem die Hooks `useEffect` zum Implementieren von Seiteneffekten, `useRef`, um Referenzen auf bestimmte Komponenten zu speichern und `useMemo` beziehungsweise `useCallback` um Werte und Funktionen zwischenzuspeichern. Neben den React-internen Hooks können auch eigene Hooks definiert werden. [12]

Wenn Komponenten zu Hierarchien kombiniert werden, so sollte die oberste Komponente, die den Zustand selbst benötigt, diesen per `useState` verwalten. Änderungen müssen (zum Beispiel über *callbacks*) an diese Komponente kommuniziert werden. Die Weitergabe der neuen Werte erfolgt dann per Parameter an die Kinder-Komponenten. Das Aktualisieren der Benutzeroberfläche vollzieht weiterhin React. [12]

Im folgenden Listing 6 wurde die Komponente `Counter` dahingehend geändert, dass der Initialwert per `props` übergeben wird. Damit kann die neue, umgebene Komponente `ResetableCounter` beim Klick auf den Button *Reset counter* den gewünschten Initialwert wiederherstellen.

```
1  function Counter(props) {
2    const count = props.count
3    const onCountChanged = props.onCountChanged
4
5    function incrementCount(event) {
6      onCountChanged(props.count + 1)
7    }
8
9    function decrementCount(event) {
10     onCountChanged(props.count - 1)
11   }
12
13   return (
14     <>
15       <p>Current count is: {count}</p>
16       <button onClick={incrementCount}> + </button>
17       <button onClick={decrementCount}> - </button>
18     </>
19   )
20 }
21
22
```

```
23  export default function ResetableCounter(props) {
24    const [count, setCount] = React.useState(0)
25    const [resetValue, setResetValue] = React.useState(0)
26
27    function countChanged(value) {
28      setCount(value)
29    }
30
31    function resetValueChanged(event) {
32      setResetValue(event.target.value)
33    }
34
35    function resetCounter(event) {
36      event.preventDefault()
37      setCount(resetValue)
38    }
39
40    return (
41      <div>
42        <form>
43          <input type="number"
44            value={resetValue}
45            onChange={resetValueChanged}
46          />
47          <button onClick={resetCounter}>Reset counter</button>
48        </form>
49        <Counter count={count} onCountChanged={countChanged} />
50      </div>
51    )
52  }
```

Listing 6: Hierarchie von React-Komponenten

`ResetableCounter` verwaltet nun den State für `count` als auch für `resetValue`. Letzterer Wert wird beim Klick auf den Button *Reset counter* in `count` gespeichert. Das `input` Textfeld ist ein *Controlled Component*[8], das den Wert von `resetValue` widerspiegelt. Zuletzt muss der neue `Counter` die Elternkomponente über Änderungen informieren sowie selbst von neuen Werten erfahren. Sobald einer der beiden Buttons des `Counter`s geklickt wird, wird letztlich `setCount` in `ResetableCounter` aufgerufen. Dies führt zu einem erneuten Zeichnen der Komponente sowie der Kind-Komponenten. Somit wird auch `Counter` neu gezeichnet. Der Wert für `count` spiegelt anschließend die letzte Änderung wieder.

Neben der Verwaltung des Zustands mittels `useState` kann auch der React-Hook `useReducer`[9] verwendet werden. Diese Funktion gibt zwei Werte zurück: Den aktuellen Wert des Zustands, wie bei `useState`, und eine *dispatch*-Funktion. Beim Aufruf der

[8] Informationen zu Controlled Components in React: https://react.dev/learn/sharing-state-between-components#controlled-and-uncontrolled-components

[9] Beschreibung zu `useReducer`: https://react.dev/reference/react/useReducer

dispatch-Funktion wird nicht direkt ein neuer Wert in dem Zustand gespeichert, sondern eine Aktion ausgelöst, die auf der *reducer*-Funktion ausgeführt wird. Der Parameter der dispatch-Funktion ist ein Objekt, welches per Konvention mindestens die Eigenschaft `type` hat. Die reducer-Funktion entscheidet dann, wie die Aktion den aktuellen Zustand verändern soll. Dazu wird in der Regel die Eigenschaft `type` der Aktion ausgewertet sowie zusätzliche übergebene Eigenschaften der Aktion verwendet. Beide Werte (der aktuelle Zustand und die Aktion) sind Parameter der reducer-Funktion. Der neue Zustand muss von der Funktion als Ergebnis zurückgegeben werden. Das Nutzen von `useReducer` erlaubt komplexere Zustände einfacher zu verwalten sowie eine einfachere Testbarkeit zu erreichen, da die reducer-Funktion vollkommen unabhängig von Komponenten und React sein kann. [12]

Das obige Beispiel aus Listing 6 ist mittels `useReducer`, wie in dem folgenden Listing 7, umzusetzen:

```
1  function Counter(props) {
2    const count = props.count
3    const dispatch = props.dispatch
4
5    function incrementCount(event) {
6      dispatch({ type: "incrementCount" })
7    }
8
9    function decrementCount(event) {
10     dispatch({ type: "decrementCount" })
11   }
12
13   return (
14     <>
15       <p>Current count is: {count}</p>
16       <button onClick={incrementCount}> + </button>
17       <button onClick={decrementCount}> - </button>
18     </>
19   )
20 }
21
22 function reducer(state, action) {
23   switch (action.type) {
24   case "incrementCount":
25     return { ...state, count: state.count + 1 }
26   case "decrementCount":
27     return { ...state, count: state.count - 1 }
28   case "updateResetValue":
29     return { ...state, resetValue: action.newResetValue }
30   case "resetValue":
31     return { ...state, count: state.resetValue }
32   default:
33     console.warn("Unknown reducer action")
34     return state
35   }
```

```
36  }
37
38  export default function ResetableCounter(props) {
39    const [state, dispatch] = React.useReducer(reducer, {
40      count: 0,
41      resetValue: 0
42    })
43
44    function resetValueChanged(event) {
45      dispatch({
46        type: "updateResetValue",
47        newResetValue: event.target.value
48      })
49    }
50
51    function resetCounter(event) {
52      event.preventDefault()
53      dispatch({ type: "resetValue" })
54    }
55
56    return (
57      <div>
58        <form>
59          <input type="number"
60            value={state.resetValue}
61            onChange={resetValueChanged}
62          />
63          <button onClick={resetCounter}>Reset counter</button>
64        </form>
65        <Counter count={state.count} dispatch={dispatch} />
66      </div>
67    )
68  }
```

Listing 7: Nutzung von `React.useReducer`

Während dieses kleine Beispiel nicht wesentlich von der Nutzung von `useReducer` profitiert, so ist die Logik in der reducer-Funktion dennoch unabhängig von den Komponenten und kann somit einfacher getestet werden. Bei umfangreicheren Komponenten kann das Zusammenführen der Behandlung des Zustands zusätzlich das Verständnis des Codes vereinfachen.

3 Konzeption

Dieses Kapitel beschreibt zunächst die Anforderungen, die sich vor und teilweise während der Entwicklung der Anwendung ergeben haben. Danach sollen die in der entwickelten Anwendung implementierten Richtlinien zur Transkription von Geheimtexten sowie ein Entwurf der Benutzeroberfläche zur Analyse dieser Geheimtexte vorgestellt werden. Der Aufbau der Geheimtexte ist elementar für die, in Kapitel 4.3 beschriebene, Verarbeitung des Geheimtextes und hat zusätzlich Einfluss auf die Konzeption der Benutzeroberfläche. Der Entwurf der Benutzeroberfläche wurde im Verlauf der Entwicklung als Vorlage für die React-Komponenten verwendet. Kapitel 4.1 zeigt Ausschnitte dieser letztendlichen Umsetzung der Oberfläche.

3.1 Anforderungen

Die folgenden Anforderungen sollen von der entwickelten Anwendung unterstützt werden:

1. Der Geheimtext muss in seine Bestandteile aufgeteilt werden, also:
   a) Trennen von Homophone mittels Text-Analyse unter Beachtung mehrerer Optionen:
      i. Grenzen zwischen Homophonen müssen anhand einer festen Länge oder eines Trennzeichen gefunden werden.
      ii. Leerraum, also Leerzeichen und Zeilenumbrüche, ist zu behandeln.
   b) Die Darstellung der einzelnen Homophone des Geheimtextes in der Benutzeroberfläche.
2. Homophone müssen auf Klartextzeichen abgebildet werden können.
3. Die Darstellung des entschlüsselten Klartextes und des gewonnenen Schlüssels muss möglich sein.
4. Der Fortschritt einer Analyse soll gespeichert und geladen werden können.
5. Eine Anpassbarkeit der Benutzeroberfläche hinsichtlich Textgröße und Farben der Elemente ist gewünscht.

6. Die manuelle Analyse soll durch Markierungen von vollständig entschlüsselten Wörtern unterstützt werden.

7. Das Hinzufügen von zusätzlichen Leerzeichen zur besseren Gliederung ist beabsichtigt.

8. Neben der Entschlüsselung sollen Homophone auch als Nomenklator oder als unsicher markiert werden können.

9. Die Auswahl von Klartextsymbolen aus mehreren, vordefinierten Alphabeten und die Definition eigener Klartextsymbole ist erwünscht.

10. Sowohl in den Geheimtext- als auch den Klartextalphabeten sollen Unicode-Zeichen unterstützt werden.

Zur Umsetzung von Anforderung 1 sind die Richtlinien „Transcription of Historical Ciphers and Keys: Guidelines“[16] des DECRYPT-Projekts zu unterstützen. Diese Richtlinien zur Einteilung sind im nächsten Kapitel (3.2) beschrieben.

3.2 Aufbau der Transkriptionen von Geheimtexten

Zunächst soll eine Beschreibung der zu unterstützenden Formatierung von Geheimtexten das Unterkapitel einleiten. Dazu werden als Erstes die typischen Darstellungen von Geheimtexten, die auf homophoner Substitution basieren, vorgestellt. Anschließend folgen die Besonderheiten der DECRYPT-Richtlinien zur Transkription.

Die einfachste Form, in der Homophone im Geheimtext vorkommen können, ist die Aneinanderreihung von Symbolen. In diesem Fall müssen die Homophone gleich lang sein. Beispielsweise soll der Geheimtext *024567*, wenn von einer Homophon-Länge von zwei Zeichen pro Symbol ausgegangen wird, innerhalb der Anwendung in die Homophone *02*, *45* und *67* aufgeteilt werden. Homophone können im Geheimtext des Weiteren mittels Trennzeichen voneinander getrennt sein. Dadurch werden unterschiedliche Längen der Homophone unterstützt. Anhand dieser Spezifikation soll der Geheimtext *418-56-123* mit dem Trennzeichen - in die Homophone *418*, *56* und *123* getrennt werden. Außerdem sind Trennzeichen mit mehreren Zeichen und verschiedene Oder-verknüpfte Trennzeichen erwünscht. So soll der Geheimtext *45-345..7-23* mit den Trennzeichen - und .. in die Homophone *45*, *345*, *7* und *23* aufgeteilt werden.

Leerzeichen, Tabs sowie Zeilenumbrüche sollen auf Wunsch in der Eingabe ignoriert oder mit in Betracht gezogen werden können. Ohne Betrachtung von Leerraum in der Eingabe, ist es angedacht Zeichen um den Leerraum zusammenzuführen, während das Betrachten von Leerraum dazu führen soll, dass die umgebenen Zeichen getrennt werden.

Wird also der Zeichenumbruch in dem Geheimtext *123\n45* entfernt und Homophone anhand der Länge 2 getrennt, so soll die Anwendung die Homophone *12*, *34* und *5* betrachten und zusätzlich eine Warnung generieren, da das letzte Homophon nicht der erwarteten Länge entspricht. Derselbe Geheimtext mit Zeilenumbrüchen muss lediglich die Homophone *12*, *3* und *45* enthalten. Die Warnung soll sich damit auf das Homophon *3* beziehen. Spezifiziert ist weiterhin, dass Leerraum optional in der Benutzeroberfläche angezeigt oder lediglich zur Aufteilung der Homophonen genutzt werden soll.

Neben diesen rudimentären Optionen sind einige syntaktische Merkmale aus den DECRYPT-Richtlinien erwünscht. Zu diesen Merkmalen gehören Metadaten, Tags und zusätzliche Markierungen an Symbolen.

Metadaten werden in den Richtlinien mit # am Zeilenanfang markiert. Vor der Raute können allerdings auch Leerzeichen oder anderer Leerraum stehen. In dem folgenden Beispiel in Listing 8 sollen zwei Zeilen mit Metadaten erkannt werden, wobei die beiden Leerzeichen vor `#CATALOG NAME` nicht Teil der Metadaten sein sollen.

```
1 #CIPHERTEXT
2 ␣␣#CATALOG␣NAME:␣Beispiel
```

Listing 8: Beispiel für unterstützte Metadaten in Geheimtexten

Tags sind in transkribierten Dokumenten mittels spitzer Klammern ausgezeichnet. Definiert werden fünf verschiedene Arten, mit eigenen Bedeutungen. Diese sind: `<SPACE>`, `<CATCHWORD>`, `<CLEARTEXT>`, `<PLAINTEXT>` und `<ABBR>`. Das Space-Tag markiert eine Stelle im Text des transkribierten Dokuments, an der von einer bewussten Wortgrenze auszugehen ist. Zeichen wurden vom Verfasser an dieser Stelle also vermutlich nicht nur zur besseren Lesbarkeit durch ein Leerzeichen getrennt, sondern der Verfasser hat (unterbewusst) die Grenzen von Wörtern im Klartext markiert. Anhand von Catchwords wird in historischen Dokumenten die Reihenfolge von Seiten kenntlich gemacht. Das Tag `<CATCHWORD Symbolname>` beschreibt ein Catchword mit dem Symbol „Symbolname“. `<CLEARTEXT DE Text>` steht für unverschlüsselten Klartext in dem Dokument. Ein solcher Klartext kann in dem originalen Dokument auch innerhalb eines verschlüsselten Abschnitts stehen. Hinter dem Namen des Tags kann ein Sprachcode stehen, der die Sprache des Klartextes identifiziert. Das *DE* steht in dem Beispiel also für einen deutschen Klartext. *Text* ist der unverschlüsselte Klartext aus dem originalen Dokument. Neben dem Cleartext-Tag gibt es das gleichermaßen aufgebaute Plaintext-Tag. Dieses beschreibt Klartext, der vom Empfänger der ursprünglichen Nachricht bereits entschlüsselt wurde. Beispielsweise wurde ein einzelnes, verschlüsseltes Wort in einem historischen Dokument mit der Entschlüsselung vom Empfänger überschrieben. Zuletzt gibt es das Abbr-Tag, welches für Abkürzungen steht. Dieses Tag kann innerhalb von Cleartext- oder Plaintext-Tags vorkommen und dient dazu Abkürzungen auszuschreiben. Da in der Transkription der originale, im Dokument vorkommende Text niedergeschrieben sein sollte, kann das Abbr-Tag als Ergänzung genutzt werden. [16]

Zeichen mit zusätzlichen Markierungen auf dem Kopf (3̇) können mittels Zahl ^Markierung (zum Beispiel 3 ^.) beschrieben werden. Das Dokumentieren von Markierungen unter Symbolen (3̣) geschieht durch Zahl_Markierung. Unterstrichene Zahlen (3̲) können mittels zweier Unterstriche hinter dem Symbol markiert werden (3___). Des Weiteren ist ein Versehen von unsicheren Symbolen mit einem dahinterstehenden Fragezeichen möglich. Zum Beispiel wird 3? notiert, sofern im Geheimtext eine 3 oder 9 geschrieben stehen könnte oder 0/6? wenn das Symbol entweder eine 0 oder 6 sein könnte. Eine unsichere Abbildung zeichnet sich dadurch aus, dass nicht vollständig gewiss ist, ob ihr Wert tatsächlich eine valide Interpretation des ursprünglichen Homophons ist. [16]

3.3 Unterstützung der DECRYPT-Richtlinien in der Anwendung

Die Tags der Richtlinien sollen nicht nur von der Anwendung erkannt, sondern auch sinnvoll innerhalb der Benutzeroberfläche präsentiert werden. So ist gewünscht, dass Tags eindeutig erkennbar sind und ihr Inhalt optisch aufbereitet ist. Während Space-Tags in der Oberfläche als Leerzeichen direkt zwischen den umgebenen Symbole erscheinen können, ist vorgesehen, dass der Inhalt der anderen Tags als solches in der Oberfläche erscheint. Die Erweiterung innerhalb eines Abbr-Tags soll die originale Abkürzung derweil ersetzen.

Die Markierungen an Symbolen in den Richtlinien sollen zunächst nicht in der geplanten Anwendung unterstützt werden, sind allerdings sinnvolle Erweiterungen für zukünftige Entwicklungen. Dank Unicode-Unterstützung der Anwendung können die Kombinationen aus Symbolen und hoch oder tief gestellten Markierungen mittels Grapheme-Clustern umgesetzt werden, um so die Symbole möglichst nah am Original zu repräsentieren. Die Markierung von unsicher transkribierten Symbolen ist außerdem in der Benutzeroberfläche direkt möglich. Unabhängig von diesen möglichen Erweiterungen, stehen die zuletzt beschriebenen syntaktischen Merkmale, einer Analyse von Dokumenten mit der konzipierten Anwendung nicht im Weg. Mittels Kenntnis der syntaktischen Regeln der DECRYPT-Richtlinien können auch diese Darstellungen erkannt werden. Innerhalb der Oberfläche kann so zum Beispiel 3^. innerhalb eines optischen Elements dargestellt werden, sofern die Separierung von Symbolen im Geheimtext durch Trennzeichen aktiviert ist.

3.4 Benutzeroberfläche

Die Anforderungen für die Benutzeroberfläche haben sich zunächst aus einigen, bereits identifizierten Schwächen des *homophonen Substitutionsanalysators* von CT2 ergeben. Es

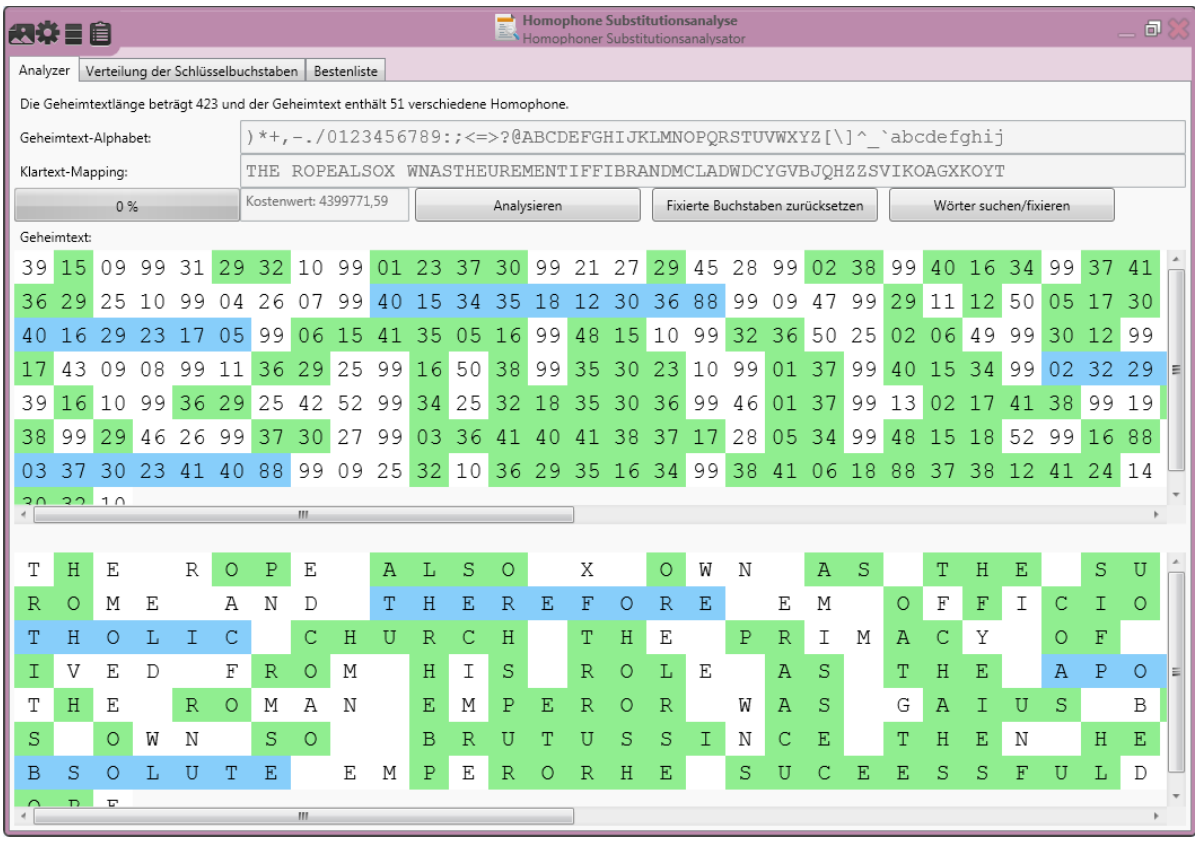

Abbildung 4: Darstellung der Homophone und deren Abbildungen in CT2

werden aktuell keine Nomenklatoren unterstützt und die Abbildung von Homophonen auf Klartextzeichen setzt eine feste Länge der Zeichen oder bestimmte, einzelne Trennzeichen voraus. Eine der größten festgestellten Schwächen stellt die optische Zuordnung von Geheimtextzeichen zu Klartextzeichen bei größeren Texten dar. Beide Texte werden getrennt voneinander dargestellt. Zugehörige Zeichen sind relativ weit voneinander entfernt und daher schwer in Kontext zu setzen. Abbildung 4 zeigt die aktuelle Darstellung in CT2.

Das Layout der Benutzeroberfläche der neuen Anwendung in CTO soll somit auf einer zeilenweisen Darstellung des Geheimtextes und Klartextes basieren. Abbildung 5 zeigt einen ersten Entwurf, wie dieser Teil der Benutzeroberfläche aussehen kann. Der obere Teil einer Zeile stellt die getrennten Homophone und der unterer Teil die zugehörigen Klartext-Symbole dar. Zur eindeutigen Zuordnung sind die Symbole in einzelnen, visuell hervorgehobenen Boxen dargestellt, deren Breite synchronisiert ist. Somit ergibt sich eine eindeutige Zuordnung von Symbolen des Geheimtextes und des Klartextes in den Spalten einer Zeile. Die Zeilen werden ebenfalls optisch hervorgehoben. Zur besseren Identifizierbarkeit sind Indizes für die Zeilen und Boxen angedacht.

Abbildung 5: Entwurf der Zeilendarstellung

Im Allgemeinen soll die Oberfläche einfach zu bedienen und dennoch leistungsfähig genug sein, um auch professionelle Arbeit zu unterstützen. Außerdem ist eine angenehme Darstellung und Nutzung sowohl auf Desktop- als auch mobilen Betriebssystemen erstrebenswert. Der Fokus soll allerdings primär auf der komfortablen Nutzung zur Analyse von Geheimtexten auf Desktop-Systemen mit größeren Displays liegen. Zur einfachen Nutzung ist sowohl die Navigation per Maus (oder Touch) als auch per Tastatur erwünscht. Des Weiteren sollen Hilfestellungen den Nutzer assistieren. Solche Hilfen bestehen unter anderem aus Warnungen bei unerwarteten Eingaben im Geheimtext.

Mit diesen initialen Ideen zur Umsetzung, wurde zunächst ein Entwurf der Benutzeroberfläche (siehe Abbildung 6) erstellt. Der Entwurf deutet das Design und Layout der bisherigen CTO-Anwendungen mittels einfacher Rechtecke und flächiger Farben an. Unter anderem zu erkennen ist der dunkelgrüne CTO-Header oben, der Anwendungs-Header darunter und die graue Anwendungsliste links im Entwurf. Unter dem Anwendungs-Header dargestellt sind Buttons für die Ein- und Ausgabe von Geheim- und Klartext sowie für dessen Optionen und ein Button zum Starten der (halb-)automatischen Analyse. Mittig sind die wichtigsten Elemente zu sehen, die Zeilen mit den Homophonen und deren Abbildungen. Neben dem schon beschriebenen, grundsätzlichen Aufbau der einzelnen Zeilen, ist die farbige Markierung von Nomenklatoren in rot und von unsicheren Symbolen in gelb sowie von markierten Wörtern (in hellblauen Boxen) ersichtlich. Die dunkelblaue Box im Entwurf stellt das aktuell ausgewählte Symbol dar. Dessen Auswahl öffnet die Software-Tastatur, die unten im Entwurf zu sehen ist. Diese dient zur Eingabe von Abbildungen des ausgewählten Homophons.

Anhand des Entwurfs konnte frühes Feedback eingeholt werden. Unter anderem stellt sich die Größe der Darstellung der einzelnen Symbol-Boxen als kritisch heraus. Bei zu großer Darstellung fehlt die Übersicht über den Kontext des Geheimtextes und dessen Entschlüsselung. Bei zu kleiner Darstellung leidet die Lesbarkeit und allgemeine Bedienbarkeit der Oberfläche. Des Weiteren sind Popups für die Auswahl der Abbildungen zu Homophonen ungünstig, da sie beim Editieren des Geheimtextes zu vielen Bewegungen in der Oberfläche führen. Die Darstellung eines festen Abschnitts, ähnlich einer Software-Tastatur auf mobilen Betriebssystemen, wurde daher präferiert.

Neben dem Entwurf wurden während der Entwicklung regelmäßige Meetings genutzt um schnelle Rückmeldung zu dem aktuellen Fortschritt der Benutzeroberfläche zu gewinnen. Viele der folgenden Anforderungen haben sich anhand dieser Meetings ergeben.

Wie in Kapitel 3.2 beschrieben, ist die Unterstützung von Homophonen mit fester, als auch unterschiedlicher Länge gewünscht. Dabei ist es angedacht, dass Unicode Grapheme-Cluster als ein Zeichen erscheinen. Tags und Metadaten aus den DECRYPT-Richtlinen zur Transkription sollen erkannt werden. Ebenfalls sollen Nomenklatoren und Nullen erkenntlich sein und es Möglichkeiten geben, Abbildungen von Homophonen zu markieren, deren Korrektheit nicht sicher ist. Um die Abbildung der Homophone zu erleichtern, soll es vordefinierte Alphabete geben, eigene Abbildungen gespeichert und das Eingeben von Abbildungen direkt mittels Tastatur unterstützt werden. Unabhängig der Zeilenlänge des Geheimtextes ist ein Umbrechen der dargestellten Zeilen in der Oberfläche, anhand der Breite des Bildschirms, angedacht. So soll ein horizontales Scrollen nicht nötig sein. Zur weiteren Steuerung der Leerraum-Zeichen und Zeilenumbrüche können Einstellungen vorgegeben werden, die beide Arten von Zeichen in der Eingabe einbeziehen oder ignorieren.

Neben diesen hauptsächlichen Funktionen, gibt es noch weitere Ideen, welche die Analyse in der Benutzeroberfläche vereinfachen würden. So werden bestimmte Symbole, wie Sternzeichen, in den Transkriptionen mitunter als Text umgesetzt. Wird das Sternzeichen Zwillinge (englisch *gemini*) als Wort *gemini* in der Transkription des Geheimtextes gefunden, so wäre eine Umsetzung in das Unicode Symbol für Zwillinge (♊) wünschenswert. Eine weitere Idee ist die Anordnung von Klartext-Abbildungen anhand ihrer Wahrscheinlichkeit. Mittels Sprachmodellen kann so erkannt werden, welcher Buchstabe zu dem aktuell ausgewählten Klartext-Symbol in dem umgebenen Kontext am wahrscheinlichsten passt. Beispielsweise sollte bei der Auswahl des Symbols „X“ in dem Text „Xaus“ zunächst die Buchstaben „H“ oder „M“ vorgeschlagen werden. Zuletzt bietet ein direktes Editieren von Homophonen in der Hauptansicht der Anwendung weitere Vereinfachungen für Nutzer. So muss bei erkannten Fehlern in der Transkription nicht die gesamte Texteingabe nach dem fehlerhaften Homophon durchsucht werden. Um die bestmögliche Implementierung der wichtigsten Anforderungen gewährleisten zu können, wurden diese Ideen auf spätere Erweiterungen verschoben.

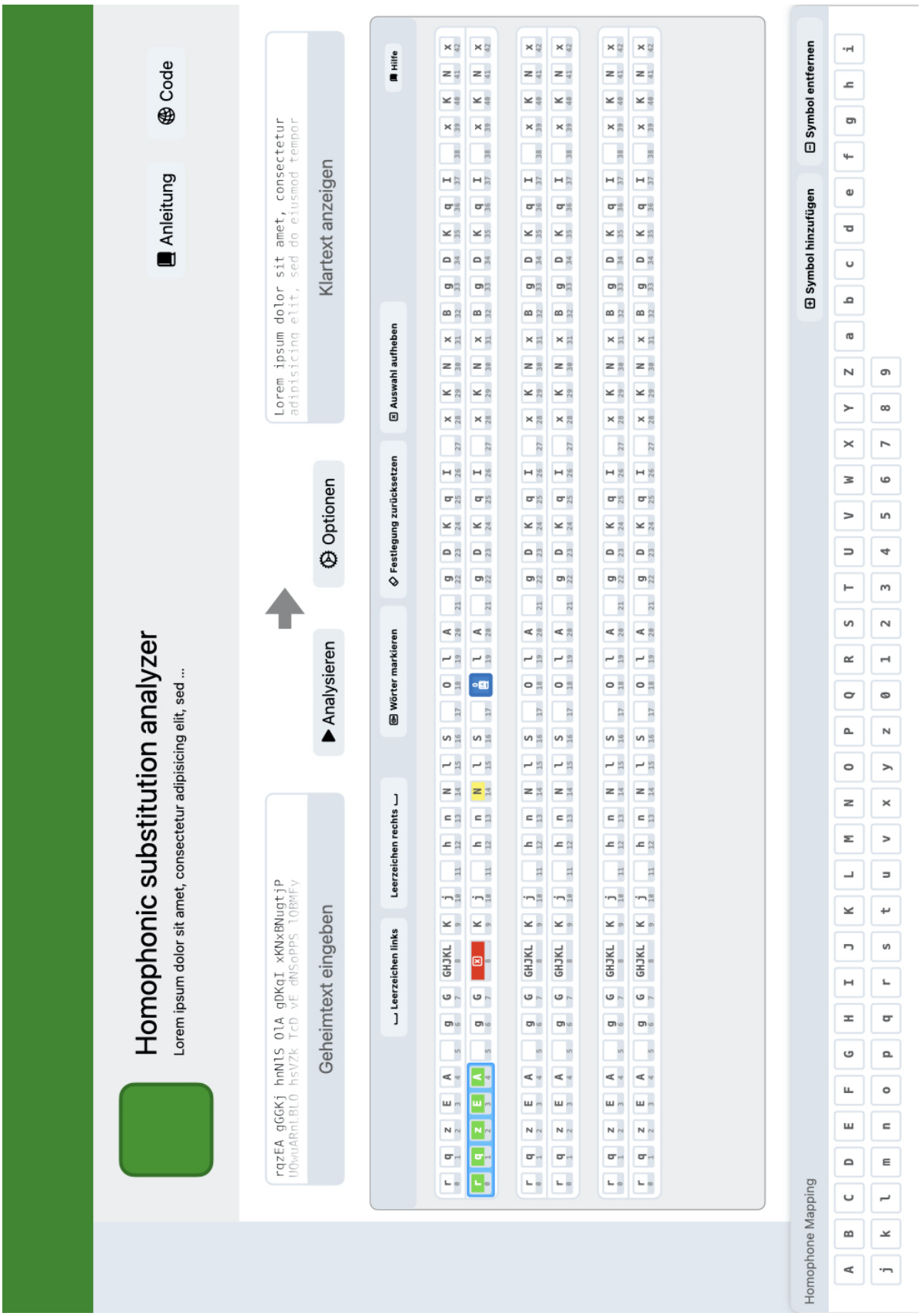

Abbildung 6: Entwurf der Benutzeroberfläche

4 Implementierung

Das folgende Kapitel erläutert einige Details der Implementierung. Nach einer Beschreibung der Benutzeroberfläche, soll der grundsätzliche Aufbau der Anwendung dargestellt und anschließend ein Blick auf die Implementierung des Geheimtext-Parsers geworfen werden. Der Aufbau der Anwendung dient dazu, neben einer Übersicht über den Code, den Datenfluss zu veranschaulichen. Im Aufgabenbereich des Parsers liegt die Verarbeitung des Geheimtextes, die zu den elementaren Anforderungen der Anwendung gehört. Anschließend erfolgt ein kurzer Blick auf die automatische Analyse und der erörterten Ansätze zur Nutzung des bestehenden Analyse-Algorithmus aus CT2. Da der Fokus der Anwendung auf der manuellen Analyse liegt, soll diese Beschreibung lediglich eine Einführung für mögliche Erweiterungen geben. Zuletzt sollen die Umsetzung des Responsive-Designs und die implementierten Performance-Optimierungen das Kapitel abschließen und zeigen wie die gewünschte Bedienbarkeit der Anwendung sichergestellt werden kann.

4.1 Umsetzung der Benutzeroberfläche

Die in Kapitel 3.4 festgelegten Hauptanforderungen wurden vollständig in der Implementierung umgesetzt. Die weiterführenden Ideen sind als solches noch nicht Teil der Umsetzung, wären aber durchaus als zukünftige Erweiterungen denkbar.

Im Folgenden soll genauer über die Umsetzung der hauptsächlichen Anforderungen geschrieben und einige Besonderheiten hervorgehoben werden. Dabei folgt die Ausarbeitung dem Ablauf der Nutzung der Anwendung, beginnend bei der Eingabe des Geheimtextes bis zum Export des Ergebnisses. Anhand von Screenshots sollen Eindrücke von dieser Nutzung vermittelt werden.

Nach der Navigation zur Anwendung innerhalb der CTO-Seite, wird die Startseite eines Projekts angezeigt. Ein Projekt bezeichnet die Analyse eines Geheimtextes mit dem aktuellen Fortschritt hinsichtlich entschlüsselter und markierter Symbole. Auf der Startseite ist entweder die Eingabe des Geheimtextes mithilfe des Geheimtext-Editors oder das Laden eines vorhandenen Projektes möglich. Sollte ein Projekt geöffnet werden, erscheint ein Fenster vom Betriebssystem, das die Auswahl von bestehenden Dateien ermöglicht.

Soll ein neuer Geheimtext zur Analyse eingegeben werden, öffnet sich der Geheimtext-Editor in einem Dialog-Fenster innerhalb der Webseite. In dem Editor ist die Eingabe des Geheimtextes mitsamt Anpassung der Eigenschaften möglich. Zu den Eigenschaften gehören das Trennverfahren für Homophone, die Behandlung von Leerraum und Zeilenumbrüchen sowie der Erkennung von Tags im Geheimtext. Das Trennverfahren für

Homophone spezifiziert, ob die Einteilung der Zeichen in der Eingabe anhand fester Länge oder eines Trennzeichens erfolgt. Werden Fehler erkannt, wie zum Beispiel die falsche Länge von einzelnen Homophonen oder unbekannte Tag-Namen, so gibt es eine Markierung mittels Warndreieck sowie ein Feld zur Auflistung und Erklärung der gefundenen Fehler.

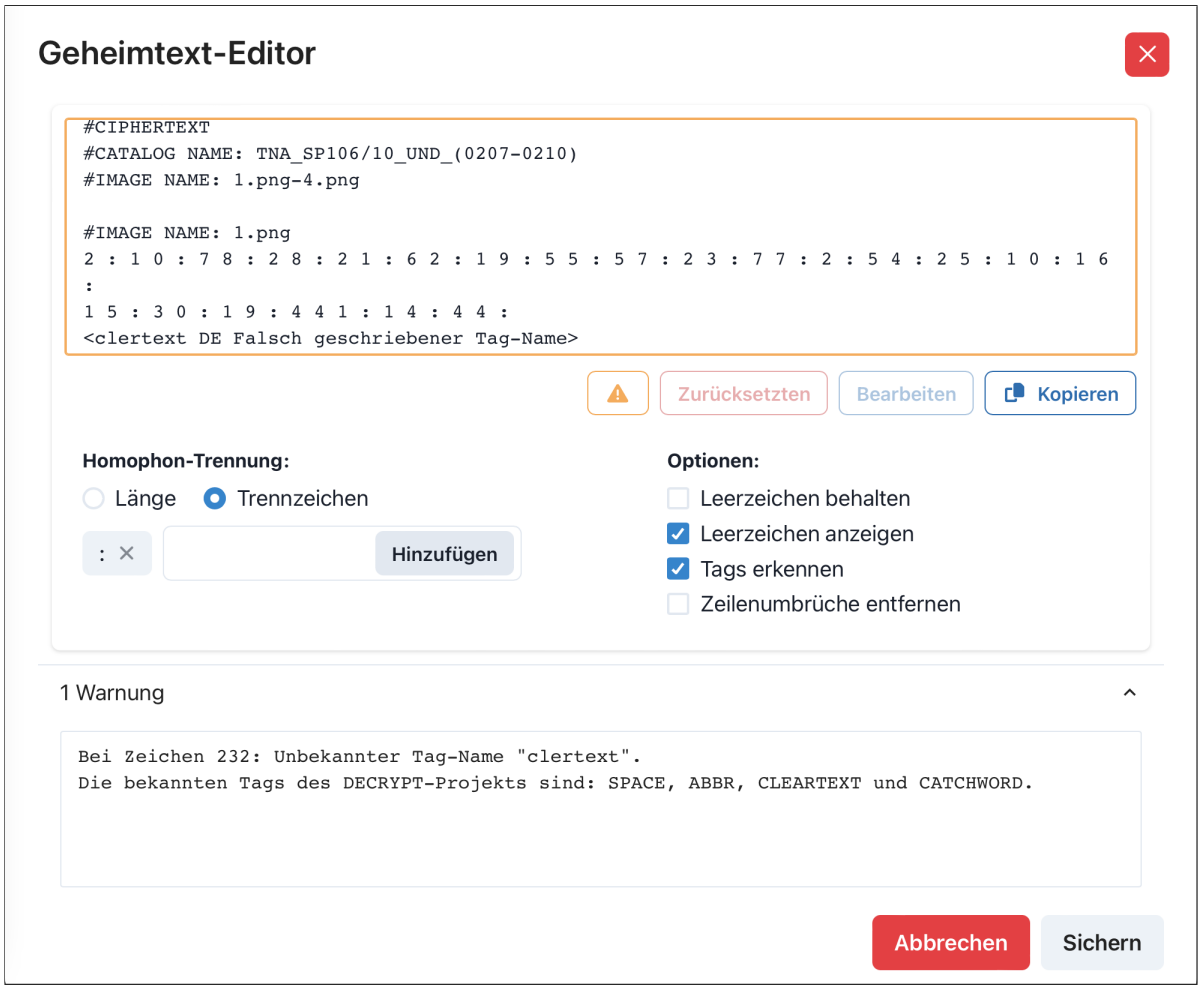

Abbildung 7: Geheimtext-Editor mit fehlerhafter Eingabe

Abbildung 7 zeigt die Eingabe eines Geheimtextes mit Metadaten und einem falsch geschriebenen Tag-Namen. Die Homophone des Geheimtextes sind anhand eines Doppelpunkts als Trennzeichen aufgeteilt. Auf Grund des Tippfehlers in dem Tag-Namen wird sowohl ein Warndreieck als auch eine ausführlichere Beschreibung in dem unteren Textfeld dargestellt.

Die Buttons zum *Bearbeiten* und *Zurücksetzen* unterhalb des Geheimtext-Textfeldes bieten verschiedene Möglichkeiten zum Durchführen von Änderungen am Geheimtext des Projekts. Mittels *Bearbeiten* ist es möglich den Geheimtext anzupassen, ohne bereits eingegebene Änderungen zu verlieren. Vorherige Abbildungen von Homophonen auf

Klartext-Symbole werden so zum Beispiel behalten. Der Button *Zurücksetzen* verhält sich dahingehend, dass der Geheimtext im Textfeld gelöscht wird und alle zu dem Geheimtext gehörenden Abbildungen, Markierungen, etc. ebenfalls zurückgesetzt werden. Um ein versehentliches Zurücksetzen zu vermeiden und zusätzlich genauere Informationen zum Verhalten zu geben, wird vor dem Zurücksetzen ein Warndialog präsentiert.

Nach dem Abschließen der Eingabe, wird die Hauptansicht dargestellt. Sie besteht aus der, bereits beschriebenen, Zeilenansicht, welche aus den gefundenen Homophonen aufgebaut ist. Diese Ansicht orientiert sich an dem Entwurf, wie er in den Anforderungen (in Kapitel 3.4) beschrieben wurde. So sind einzelne Zeilen deutlich voneinander abgehoben. Jede Zeile ist durch eine fortlaufende Zeilennummer markiert. Leere Zeilen in der Eingabe oder solche, die lediglich Metadaten enthalten, werden in der Oberfläche ausgelassen. Daneben werden Zeilen umgebrochen, sofern die Darstellung der zugehörigen Eingabezeile die Breite des Bildschirms überschreitet. In diesem Fall wird eine Ellipse als Index der Zeile dargestellt. Um die gewünschte Übersichtlichkeit des gesamten Dokuments bei der Analyse des Geheimtextes zu behalten, wird den Symbolboxen eine maximale Breite zugeordnet. Beim Überschreiten dieser Breite, zeigt die Box zunächst eine Ellipse und bietet die Möglichkeit entweder einen Tooltip anzuzeigen oder die Box zu vergrößern, sobald der Cursor über der Box schwebt. Sowohl die Größe, als auch die Art der Enthüllung des abgekürzten Symbols, kann innerhalb der Darstellungsoptionen angepasst werden. Diese Einstellungen sind innerhalb des Projekt-Menüs aufzufinden.

Das Projekt-Menü befindet sich in einer schwebenden Leiste, die zusätzlich Aktionen zum Ansehen des entschlüsselten Klartextes, zum Markieren von vollständig entschlüsselnden Wörtern, zum Entfernen von gesetzten Abbildungen und vielem mehr anbietet. Diese Aktionsleiste ist immer oberhalb der Zeilenansicht zu sehen, unabhängig davon, welcher Ausschnitt des Dokuments betrachtet wird.

Sollte der Parser Fehler im Geheimtext gefunden haben, so werden diese ebenfalls direkt an den Symbolboxen dargestellt. Dafür dient eine orangene Markierung unterhalb der betreffenden Box. Diese soll an Markierungen von Compiler-Fehlern in IDEs oder Rechtschreibfehlern in Texteditoren erinnern. Ein Tooltip an der Markierung zeigt zusätzlich den zugehörigen Fehlertext, mit ausführlichen Informationen zu dem Fehlergrund. Neben unerwarteten Längen von Homophonen können so zum Beispiel auch unbekannte Tag-Namen oder nicht geschlossene Tags angemerkt sein.

Die dargestellten Symbole in Abbildung 8 basieren auf der vorherigen Eingabe aus Abbildung 7. Die leeren Zeilen sowie Zeilen mit Metadaten sind nicht Teil der Darstellung. In der letzten Zeile ist eine orangene Markierung des fehlerhaften Tag-Namens zu erkennen. Die Aktionsleiste oberhalb der Symbole bietet neben dem Projektmenü Zugriff auf den Geheimtext-Editor und die Klartext-Ausgabe. Um die anderen Aktionen ausführen zu können, muss eine Auswahl getroffen werden. Links im Projekt-Menü ist mittels eines Symbols der ungesicherte Zustand des Projekts markiert.

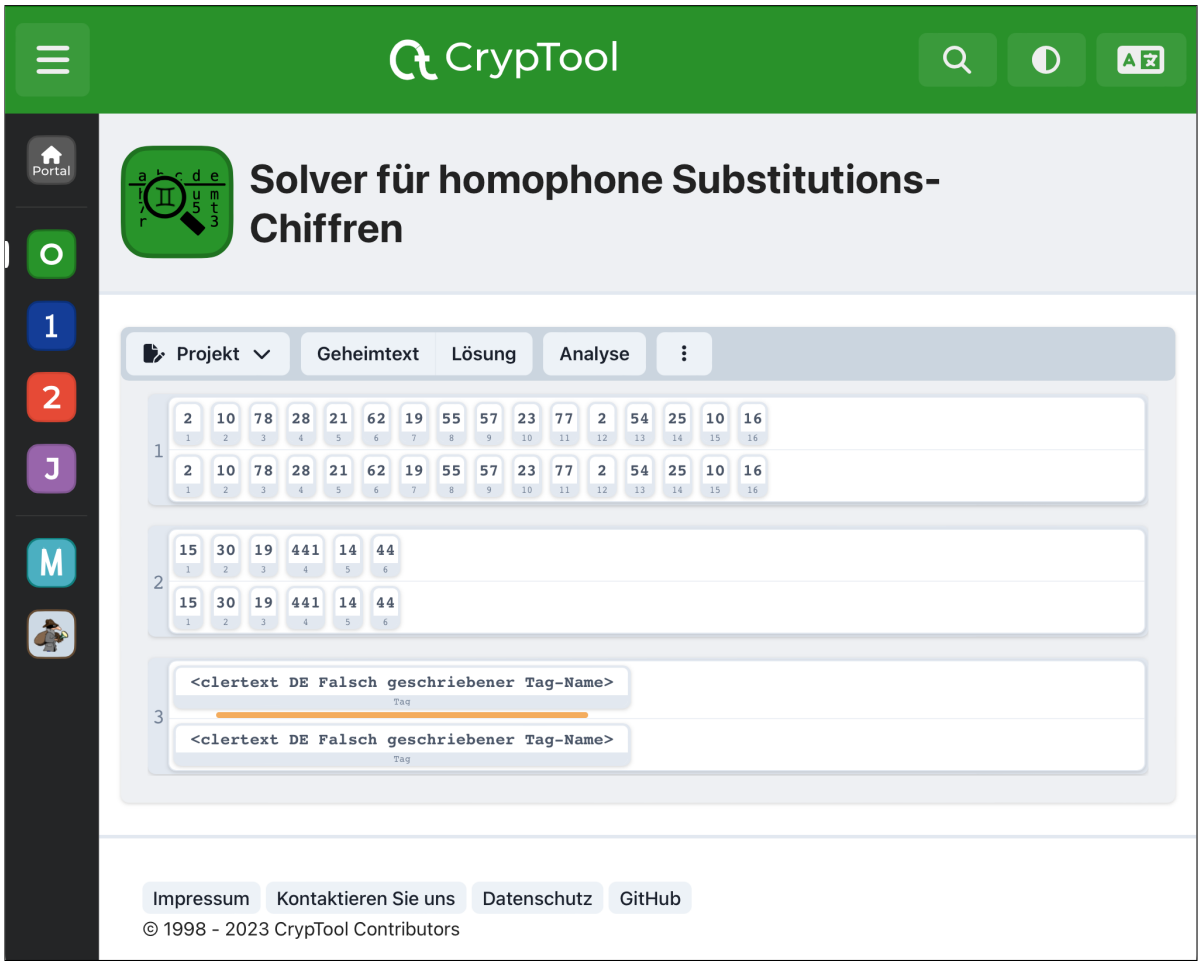

Abbildung 8: Hauptansicht mit fehlerhaftem Tag-Namen

Beim Bewegen der Maus über einzelne Symbole werden die beiden Boxen in einer Spalte zur einfacheren Orientierung hervorgehoben. Beim Anklicken einer der beiden Boxen ändert sich die Hervorhebung von einer grauen zu einer blauen Farbe und die Software-Tastatur zur Abbildung der markierten Homophone öffnet sich am unteren Bildschirmrand. Eine Markierung von Symbolboxen der weiteren Vorkommen desselben Homophons im Geheimtext dient zusätzlich zur besseren Übersichtlichkeit bei der Analyse. Dazu werden die entsprechenden Symbolboxen mit einem blauen Rand markiert.

Die Software-Tastatur unterstützt hauptsächlich bei der Abbildung von markierten Homophonen zu Klartext-Symbolen. Standardmäßig bietet sie zur Abbildung Zahlen sowie großgeschriebene Buchstaben an. Die vordefinierten Symbole sind in Alphabete geordnet, die durch ein Dropdown-Menü im Header der Tastatur gewählt werden können. Zusätzlich bietet die Software-Tastatur die Möglichkeit, weitere Symbole per Textfeld hinzuzufügen. Neben den Tasten zur Abbildung von Homophonen auf Symbole, zeigt die Komponente auch einige Zusatzinformationen im Header, die vor allem bei der Nutzung

mobiler Betriebssysteme hilfreich sind. So wird zum einen eine Beschreibung der Abbildung dargestellt. Diese erlaubt es auch bei kompakter Darstellung der Symbolboxen den vollständigen Text der Homophone und deren Abbildungen lesen zu können. Zum anderen gibt es Pfeiltasten zur schnelleren Navigation zum vorherigen oder folgenden Homophon.

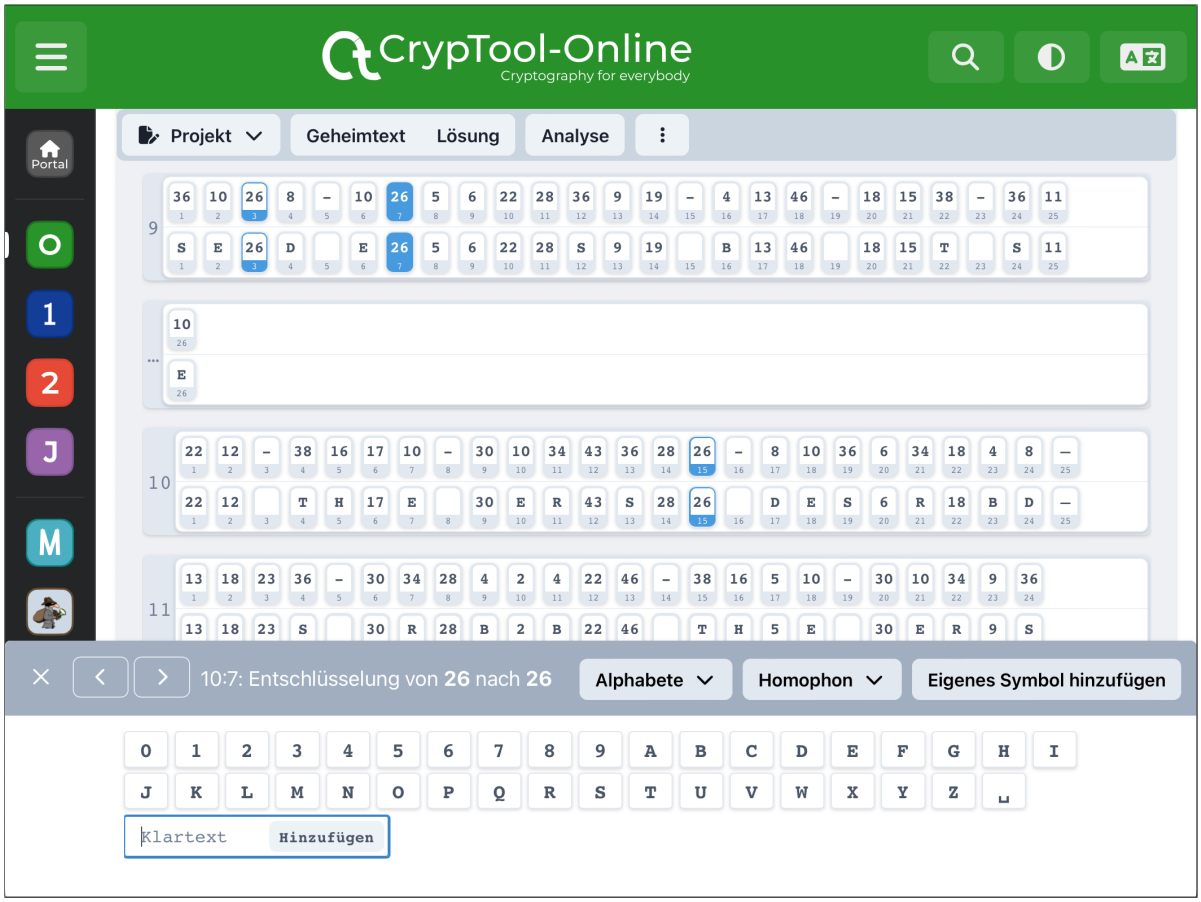

Abbildung 9: Hauptansicht mit geöffneter Software-Tastatur

Abbildung 9 zeigt die unterschiedlichen Markierungen von Symbolen in der Hauptansicht. Das aktuell markierte Homophon ist in Zeile 9 Spalte 7 und hat den Wert *26*. Es wird durch eine durchgehende blaue Markierung hervorgehoben. Die weiteren Vorkommen sind mit einem blauen Rand hervorgehoben. In dem Beispiel ist zusätzlich die Zeile 9 der Eingabe in eine zweite Zeile umgebrochen. Die geöffnete Software-Tastatur bietet, neben den Tasten für die Standard-Abbildungen und dem bereits eingegebenen Leerzeichen für das Homophon „-“, das Eingabefeld für eigene Abbildungen an. Wird eine Eingabe mit dem Zeilenschalter oder dem Button „Hinzufügen“ bestätigt, wird das aktuell markierte Homophon auf die Eingabe abgebildet. Außerdem wird diese Eingabe auch in der Software-Tastatur für weitere Verwendungen gespeichert.

Um eine schnellere Bedienung per Tastatur zu ermöglichen, kann bei markiertem Homophon ein Buchstabe auch direkt auf der Hardware-Tastatur eingetippt werden. Außerdem ist es möglich, per Zeilenschalter das Textfeld zur Eingabe eigener Symbole in der Software-Tastatur zu öffnen. So können komplexere Symbole ebenfalls schnell, ohne Nutzung der Maus, eingetragen werden.

Nomenklatoren, Nullen und unsichere Abbildungen können in der Oberfläche mittels Checkboxen im Homophon-Menü der Software-Tastatur markiert werden. Nomenklatoren und Nullen erhalten eine rote Markierung in der Zeile der Klartext-Abbildungen sowie je ein spezielles Symbol. In der generierten Klartext-Ausgabe steht allerdings weiterhin der Geheimtext für den Nomenklator. Unsichere Abbildungen werden mittels gelber Farbe als Hintergrund der Klartext-Symbole markiert.

Um die beschriebenen Schritte möglichst komfortabel an einem Desktop-System durchzuführen, bietet sich die Navigation per Tastatur an. Nachdem ein Homophon zur Abbildung ausgewählt wurde, kann mittels Pfeiltasten innerhalb der Zeilen und Spalten navigiert werden. Das Navigieren ist grundsätzlich dem Verhalten eines Texteditors nachempfunden. Navigation nach oben und unten über Zeilen hinweg muss die unterschiedliche Länge von Zeilen in Betracht ziehen. Liegt zwischen zwei langen Zeilen eine kürzere, so sollte der Index der Spalte auch beim Überspringen der kürzeren Zeile erhalten bleiben, selbst wenn der Index die Länge der kurzen Zeile überschreiten würde. Zusätzlich zum reinen Navigieren ist das Treffen einer Auswahl mittels Halten der Shift-Taste möglich. Die ausgewählten Homophone können dann zusammen auf dasselbe Klartext-Symbol abgebildet oder die bereits entschlüsselten Klartext-Symbole als Wort markiert werden. Neben der zusammenhängenden Auswahl ist zusätzlich auch eine unterbrochene Auswahl möglich. Das Auswahl-Verhalten empfindet sowohl Texteditoren als auch Darstellungen von Symbol-Ansichten (zum Beispiel in Dateibrowsern) nach. Sowohl bei der Navigation, als auch der Auswahl ist es wichtig, dass die Implementierung die Bewegungsrichtung der Pfeiltasten bedenkt. Andernfalls stellt sich das Scroll-Verhalten, als auch die Änderung von bereits getroffenen Auswahlen, als äußerst unerwartet heraus.

Während der Analyse des Geheimtextes gibt es jederzeit die Möglichkeit, den aktuellen Fortschritt zu speichern. Dazu befindet sich in dem Projekt-Menü, ein Button für das Speichern, welcher den Download-Prozess startet. Das Projekt wird entsprechend den Einstellungen des Browsers gesichert. Teil der Projekt-Datei sind:

- jegliche Eingaben und Änderungen am Geheimtext und dessen Einstellungen
- die Abbildungen auf Klartext-Symbole
- Markierungen von Homophonen und Wörtern
- hinzugefügte Klartext-Symbole in den Alphabeten

- in der Software-Tastatur ausgewählte Alphabete
- hinzugefügte Leerzeichen zwischen Homophonen
- Darstellungsoptionen

Um einen Verlust des Analyse-Fortschrittes zu vermeiden, wird bei ungesicherten Änderungen das Schließen des Browsertabs oder -fensters mit einer Warnung versehen. Eine mögliche Verbesserung für zukünftige Erweiterungen wäre die Nutzung der *File System Access API*[10] zum Laden und Speichern. Mittels dieser, könnte beim Speichern ein spezifischer Ort auf dem Dateisystem gewählt und bestehende Projekte überschrieben werden. Derzeit ist diese API allerdings nur in wenigen Browsern implementiert.

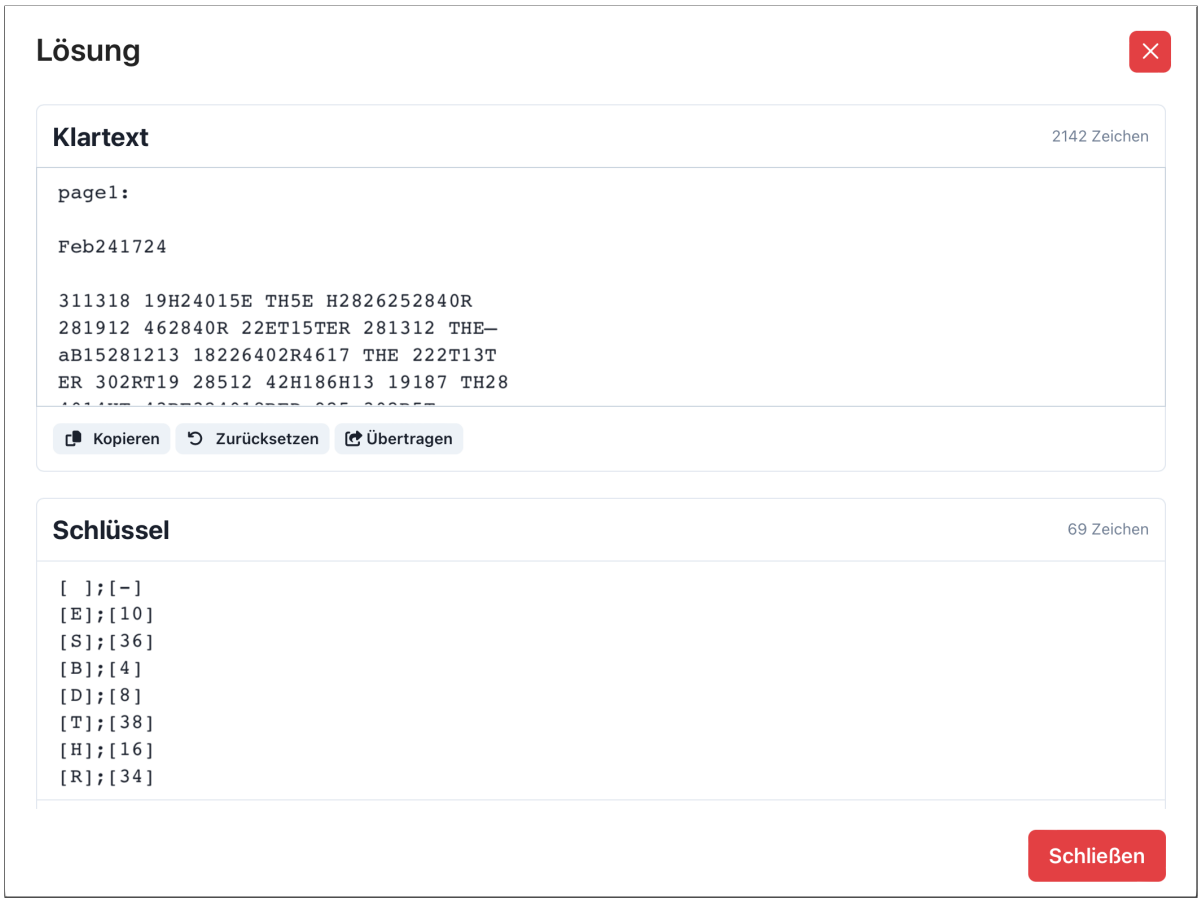

Abbildung 10: Ausgabe des aktuellen Klartextes und Schlüssels

Ist die Analyse abgeschlossen oder soll der aktuelle Fortschritt betrachtet werden, so lässt sich mittels des Buttons *Lösung* in der schwebenden Aktionsleiste ein Dialogfens-

[10] Beschreibung der File System Access API und die derzeitige Browserkompatibilität: https://developer.mozilla.org/en-US/docs/Web/API/File_System_Access_API

ter öffnen, welches neben dem Klartext und dem Schlüssel auch Buttons zum Kopieren und zum Export in andere CTO-Anwendungen anzeigt. Abbildung 10 zeigt eine beispielhafte Ausgabe einer Entschlüsselung und den dazugehörigen Schlüssel im CT2-Schlüsselformat.

4.2 Strukturierung der Anwendung

CTO nutzt *next.js*[11] als Framework für die Entwicklung der Webseite. Die eigentliche Webseite ist in viele sogenannte CTO-Anwendungen eingeteilt. Jede Anwendung erbt dabei ein vordefiniertes Layout und wird in einem Ausschnitt der Seite gerendert. Die Seitenleiste sowie der Anwendungsheader sind vordefiniert und werden um den Rahmen der CTO-Anwendung gezeichnet. Anhand von Informationen, die in einer zentralen JSON-Datei verwaltet werden, wird die CTO-Anwendung der Webseite bekannt gemacht und damit in der Anwendungsliste angezeigt und mit Informationen für den vordefinierten Header versehen.

Die Architektur der Anwendung folgt dem vorausgesetzten Aufbau durch React. Komponenten sind in einer Baum-Hierarchie angeordnet und tauschen Daten durch `props`, Callbacks oder `dispatch`-Funktionen aus. Die Hauptkomponente der Anwendung liegt im Ordner `ctoapps/homophonic-substitution-analyzer` in der Datei `HomophonicSubstitutionAnalyzerComponent.jsx`. Diese Komponente ist für die grundlegende Datenverwaltung eines geöffneten Projekts zuständig und zeigt die wichtigsten Komponenten der Benutzeroberfläche als Kinder. Zu diesen Komponenten zählen das `SymbolGrid`, welches `SymbolRow` und `SymbolBox` zur Darstellung der Zeilen und einzelnen Symbole des eingegebenen Geheimtextes nutzt, der `CiphertextEditor` zur Eingabe und Veränderung des Geheimtextes sowie das `HomophoneMappingKeyboard`, welches die Software-Tastatur zur Abbildung von Homophonen auf Klartext-Zeichen umsetzt. Der `ActionButtonHeader` schwebt am oberen Bildschirmrand über den Symbolen und bietet Zugriff auf häufig benötigte Funktionen. Wird die Anwendung initialisiert, sind zunächst lediglich die Komponenten `HomophonicSubstitutionAnalyzerComponent`, `ActionButtonHeader` und `SymbolGrid` zu sehen. Das `HomophonicSubstitutionAnalyzerComponent` ist für die Darstellung der Dialog-Fenster zuständig. Beispiele für Dialog-Fenster sind der Geheimtext-Editor oder die Darstellungsoptionen. Wird der Geheimtext-Editor geöffnet und ein Geheimtext mit den zugehörigen Optionen eingegeben, so wird das `HomophonicSubstitutionAnalyzerComponent` die Verarbeitung des Geheimtextes veranlassen und die Ergebnisse zum einen speichern und zum anderen an das `SymbolGrid` weiterreichen. Das `SymbolGrid` initialisiert mit diesen Ergebnissen die Liste der `SymbolRows`. Daneben registriert die Komponente mehrere Funktionen zur Behandlung von Auswahl- und Navigationsoperationen. Die Komponente ist darüber hinaus dafür zuständig, dass die Darstellung zur aktuellen Auswahl scrollt und Tastaturbefehle die Auswahl verän-

[11] Für mehr Informationen zu next.js siehe: `https://nextjs.org`

dern. Außerdem verwaltet das `SymbolGrid` die Darstellung der Software-Tastatur. Jede `SymbolRow` ist für die Initialisierung der `SymbolBox`en zuständig. Sie nutzt Informationen, die vom `SymbolGrid` bereitgestellt werden, um die Darstellung der Symbole anzupassen. Eine `SymbolBox` ist letztlich dafür verantwortlich die einzelnen Symbole richtig darzustellen und bei Aktionen die zuständigen Komponenten zu benachrichtigen.

Abbildung 11 zeigt eine schematische Darstellung der Hauptansicht der Anwendung, in der die einzelnen Teile der Oberfläche mit den Namen der zuständigen Komponenten versehen sind. Die in der Abbildung zu sehende Hierarchie der Komponenten entspricht auch größtenteils der Strukturierung im generierten HTML-Dokument. Lediglich das `HomophoneMappingKeyboard` ist nicht Kind des `HomophonicSubstitutionAnalyzer-Component` und wird am Ende des HTML-Dokuments angehangen.

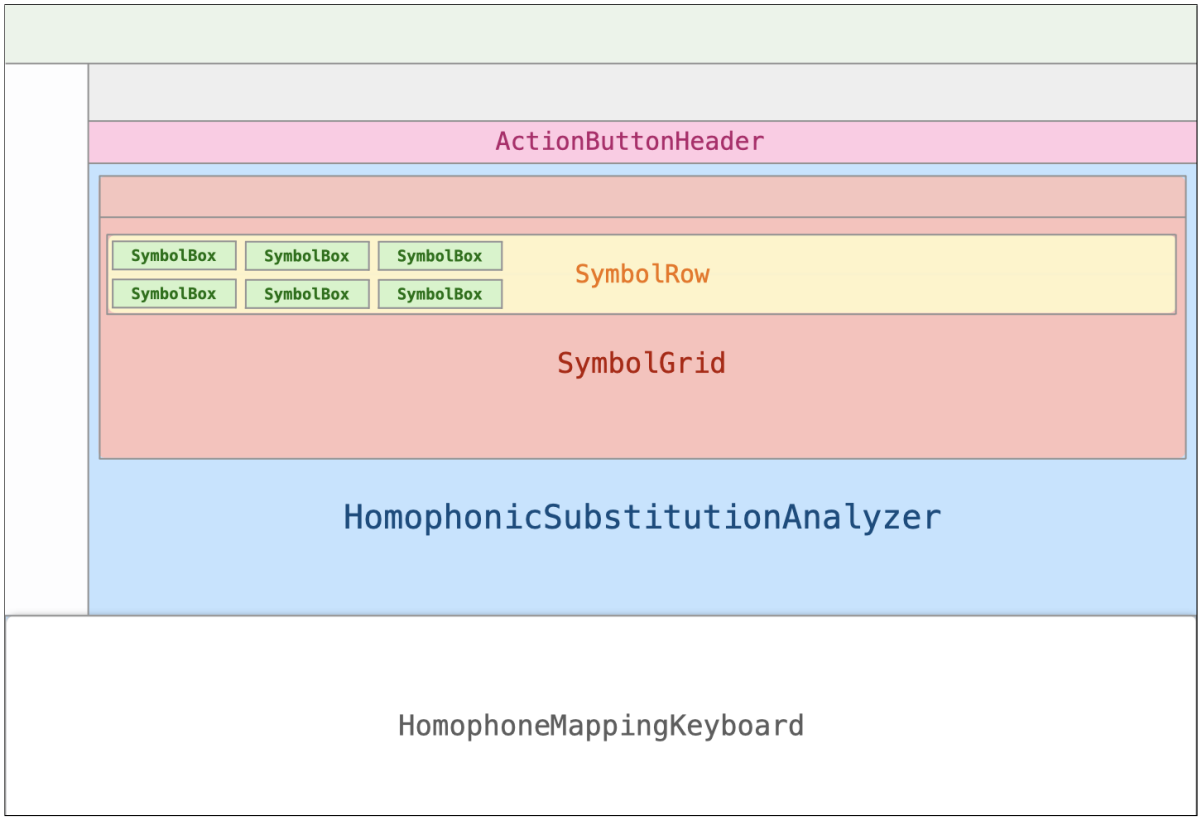

Abbildung 11: Schematische Darstellung der Hauptansicht mit den Namen der Komponenten

Wie bereits in den Grundlagen in Kapitel 2.8 beschrieben, erfolgt der Datenfluss in React von den Eltern zu den Kindern. Die meisten Komponenten nutzen den React-Hook `useState`, `props` sowie callbacks um ihre temporären Daten zur Beeinflussung der Darstellung zu speichern und zu editieren. Daten, die ein Projekt beschreiben, wie zum Beispiel der eingegebene Geheimtext, die Einstellungen, wie dieser Geheimtext zu verarbeiten ist oder die in der Benutzeroberfläche markierten Wörter, wer-

den hingegen in der Hauptkomponente der CTO-Anwendung mittels des React-Hooks `useReducer` verwaltet. Diese Daten sind nicht nur für die Darstellung und Manipulation in der Benutzeroberfläche wichtig, sondern zugleich für die Speicherung eines Projekts relevant. Neben dem `HomophonicSubstitutionAnalyzerComponent` nutzt auch die Komponente `SymbolGrid` den Hook `useReducer` zur Verwaltung ihrer Daten. Diese Daten betreffen die Auswahl und Navigation der Symbole in der Hauptansicht. Beide Komponenten nutzen `useReducer` zur einfacheren Verwaltung ihres Zustands. Anstelle von vielen callback-Funktionen, können beide Komponenten die dispatch-Funktionen der Reducer an ihre Kind-Komponenten weiterreichen. Zur Unterscheidung wird der Reducer des `SymbolGrid`s als `navigationReducer` und dessen dispatch-Funktion als `navigationDispatch` im Code benannt. Die Aufteilung in zwei Reducer dient dazu, die Aufgabenbereiche sauber zu trennen und besser zwischen persistenten und temporären Daten unterscheiden zu können. Ein weiterer Vorteil der Nutzung von Reducern ist, dass unabhängige Funktionen implementiert werden können. Dies wird im Fall des Navigation-Reducers genutzt, um die aufwändige Logik der Navigation ausgiebig zu testen.

Um die gesamte Anwendung sinnvoll zu strukturieren, werden Komponenten, Model- und Helper-Klassen, React-Hooks und Testfälle in entsprechend benannten Ordnern gespeichert. Vor allem für die unabhängige Logik der Anwendung sind Unit-Tests[12] vorhanden. So ist die Implementierung der Verarbeitung des Geheimtextes und der Navigation in der Benutzeroberfläche ausgiebig durch Tests überprüft.

4.3 Verarbeitung des Geheimtextes

Der folgende Abschnitt soll die Implementierung der Verarbeitung des Geheimtextes genauer erläutern. Innerhalb der Anwendung ist für die Erkennung der Homophone sowie der Unterstützung der DECRYPT-Richtlinien[13] der Parser zuständig.

Die grobe Struktur der Verarbeitung von Eingabetexten ist in die Bereiche des Scanners und Parsers eingeteilt. Die Aufgabe des Scanners ist es, die kleinsten syntaktischen Merkmale der Eingabe zu erkennen und diese an den Parser weiterzugeben. Anhand dieser Einheiten kann der Parser die eigentliche Struktur der Eingabe verifizieren, in die gewünschten, größeren Einheiten zusammenfassen und als Ableitungsbaum bereitstellen. Neben dem Baum können etwaige Fehler ein Teil der Ausgabe sein. Der Parser ist wiederum eine Komposition aus mehreren Klassen. Die exportierte Klasse ist ein *Decorator*[14] um den eigentlichen Parser. Innerhalb des Decorators werden Transformationen auf

[12] Weitere Informationen zu den verschiedenen Arten von Tests: https://www.atlassian.com/de/continuous-delivery/software-testing/types-of-software-testing

[13] siehe Kapitel 3.2

[14] Eine Beschreibung des Entwurfmusters bietet: https://www.heise.de/blog/Patterns-in-der-Softwareentwicklung-Das-Decorator-Muster-7315588.html

dem generierten Ableitungsbaum des eigentlichen Parse-Vorgangs implementiert. Diese Aufteilung erlaubt die Aufgaben der einzelnen Klassen in der Implementierung sauber zu trennen.

Abbildung 12 zeigt ein Klassendiagramm der für die Verarbeitung des Geheimtextes zuständigen Klassen. Um die Übersichtlichkeit zu wahren, sind nur die relevanten Details in dem Diagramm dokumentiert. Neben dem `SimpleParser` und seinem Decorator `TransformingParser`, sind der `Scanner` und die im Folge des Kapitels beschriebenen Klassen `Warning`, `Token`, `TreeTransformation`, `TreeConsistencyChecker`, `Node`, `NonTerminal` und `Terminal` dokumentiert.

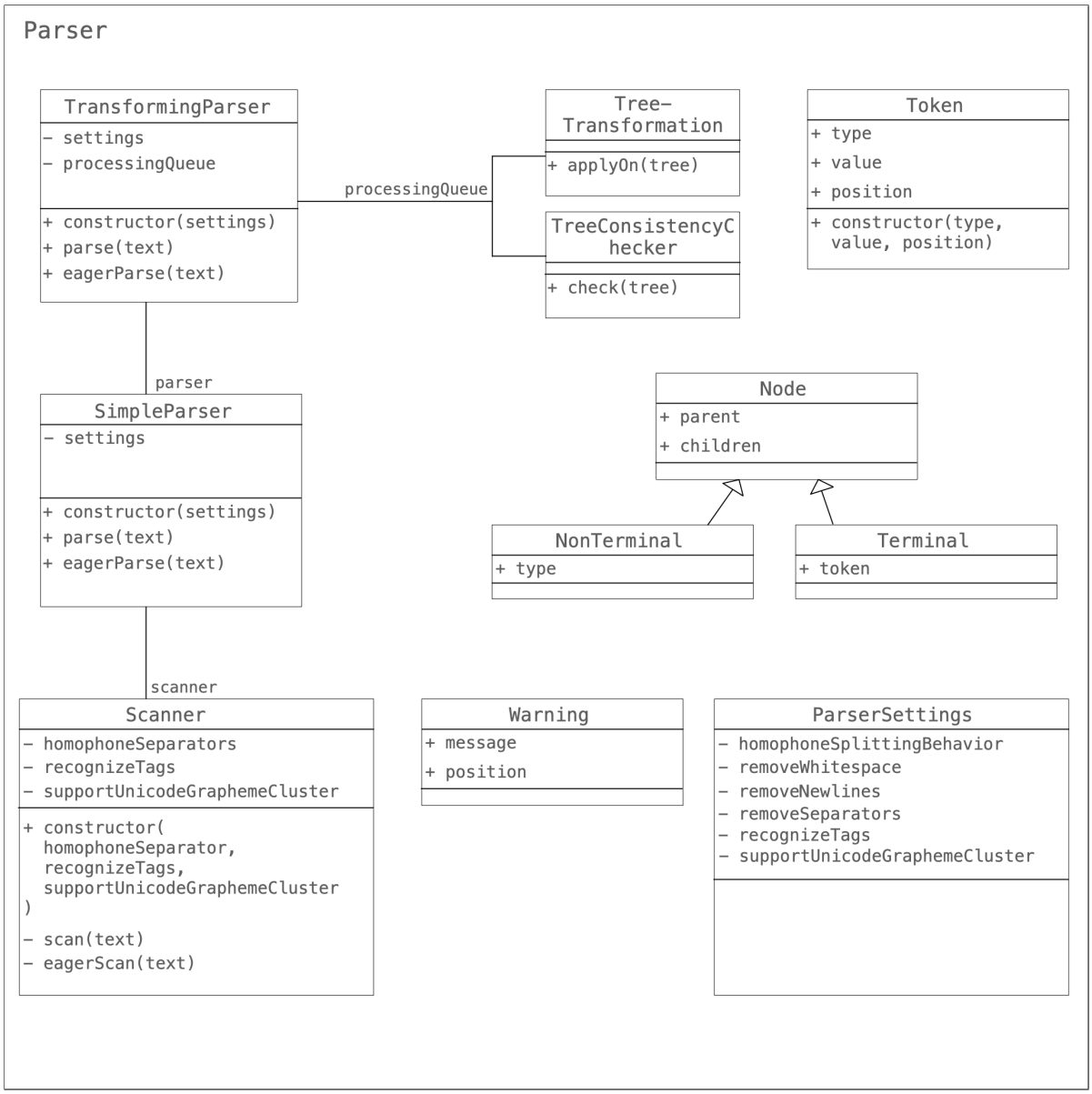

Abbildung 12: Klassendiagramm des Parsers, Scanners und der zugehörigen Klassen

Bevor der genauere Aufbau des Parsers beschrieben wird, soll zunächst der Ablauf der Verarbeitung im Scanner vorgestellt werden. Als Eingabe erhält der Scanner den gesamten eingegebenen Text. Nach der Transformation wird als Ausgabe ein Array von Token zurückgegeben. Ein Token besitzt einen Typ, der die Merkmale eines betrachteten Elements beschreibt, den Wert des Elements und dessen Position in der Eingabe. Die unterschiedenen Typen sind `WHITESPACE`, `NEWLINE`, `SEPARATOR`, `OPENING_TAG`, `CLOSING_TAG`, `METADATA` und `TEXT`. Für einen Ausschnitt der Definition der Klasse `Token` sowie der Konstanten in der Klasse `TokenType` siehe Listing 9. Der Scanner nimmt während der Verarbeitung keine Änderungen an der Eingabe vor. Der eingegebene Text kann somit, durch Serialisieren der Werte der Token, wieder vollständig rekonstruiert werden.

```
1  class Token {
2    type
3    value
4    position
5    // ...
6  }
7
8  class TokenType {
9    static WHITESPACE = "whitespace"
10   static NEWLINE = "newline"
11   static SEPARATOR = "separator"
12   static OPENING_TAG = "opening_tag"
13   static CLOSING_TAG = "closing_tag"
14   static METADATA = "metadata"
15   static TEXT = "text"
16 }
```

Listing 9: Ausschnitt der Klassen `Token` und `TokenType`

Soll der eingegebene Geheimtext anhand von Trennzeichen aufgeteilt werden, so müssen diese dem Scanner bereits bekannt sein. Nur auf diese Weise ist die korrekte Erkennung des Typs eines Token möglich. Angenommen „-“ ist in einem Geheimtext ein solches Trennzeichen zwischen Homophonen. Ohne dessen Kenntnis kann der Scanner „-“ nur als Teil eines Homophones deklarieren. Ebenfalls ist so eine Priorisierung innerhalb des Scanners möglich. Da zum Beispiel Leerzeichen als mögliche Trennzeichen erlaubt sind, priorisiert der Scanner Trennzeichen gegenüber Leerraum. Neben den Trennzeichen gibt es noch weitere Einstellungen, die dem Scanner vor der Verarbeitung mitgeteilt werden können. Zum Beispiel, ob die Tags der DECRYPT-Richtlinien unterstützt werden sollen oder eine Erkennung von Unicode Grapheme-Clustern gewünscht ist.

Um die eigentliche Verarbeitung durchzuführen, bietet der Scanner die Methoden `scan()` und `eagerScan()`. Innerhalb der Methode `scan()` erfolgt die gesamte Verarbeitung erst bei Bedarf. `eagerScan()` ruft intern lediglich `scan()` auf, bis die Eingabe abgearbeitet ist. Der Scanner nutzt das Generator-Konzept aus JS, um Fortschritte beim Scannen nur bei Nachfrage zu generieren. Dadurch ist es möglich besonders große Dokumente schrittweise zu verarbeiten. Die Benutzeroberfläche kann bei Bedarf den Parser und da-

mit auch den Scanner anweisen mehr Elemente der Eingabe zu ermitteln. Bei jeder Anfrage wird die Eingabe Zeichen für Zeichen innerhalb von `scan()` durchlaufen. Zunächst wird der Typ des Zeichens ermittelt. Dieser Test benötigt einen gewissen *Lookahead*. Für Homophon-Trennzeichen ist dieser zum Beispiel die Länge des Trennzeichen - 1. Zum Erkennen von Metadaten muss hingegen bis zum Zeilenende vorausgeschaut werden. Der Wert des aktuellen Zeichens kombiniert mit dem Lookahead wird mit einem Muster verglichen. Ist der Wert gleich einem der bekannten Trennzeichen, handelt es sich zum Beispiel um den Typ `SEPARATOR`. Anhand des erkannten Typs wird dann das passende Token erzeugt. Der Typ des Zeichens, sein Wert und die Position werden eingetragen. Der Wert entspricht dem aktuellen Zeichen, zusammen mit allen Zeichen des Lookahead. Die Position besteht aus dem Index des aktuellen Zeichens und der gesamten Länge des aktuellen Zeichens und dem Lookahead. Für alle Typen außer `TEXT` wird das Token nun per `yield` an den Aufrufer übermittelt. Text-Token werden so lange gesammelt, bis ein anderer Typ angetroffen wird. Nach jedem Token springt der Scanner über *Lookahead* Zeichen.

Beispielsweise generiert die Eingabe des Textes *123456*, wie im Listing 10 zu sehen ist, nur ein Token vom Typ `TEXT` und Position 0 mit Länge 6. Zum Scannen wird die Methode `eagerScan()` verwendet, um direkt ein Array als Resultat zu erhalten, ohne zunächst einen Iterator umwandeln zu müssen.

```
1  const input = "123456"
2  const scanner = new Scanner({ homophoneSeparator: null })
3  scanner.eagerScan(input) // => [{"position": [0, 6], "type": "text", "
     value": "123456"}]
```

Listing 10: Aufteilung einer einfachen Eingabe durch den `Scanner`

Der Text „123: 4“ in Listing 11 wird anhand des Trennzeichens „:“ in vier Token eingeteilt. Nach dem ersten Token vom Typ `TEXT` und Wert „123“ und dem zweiten Token vom Typ `SEPARATOR` und dem definierten Trennzeichen, folgt ein Token mit einem Leerzeichen als Wert und Typ `WHITESPACE`. Abschließend endet die Ausgabe mit einem Token vom Typ `TEXT` und Wert „4“. Die Positionen, repräsentiert als Startindex und Länge, entsprechen dem Vorkommen der Werte in dem Eingabestring. Mittels `output.map(token => token.value).join()` kann die Eingabe vollständig rekonstruiert werden.

```
1  const input = "123: 4"
2  const scanner = new Scanner({ homophoneSeparator: [":"] })
3  const output = scanner.eagerScan(input)
4  /* output => [
5       { position: [0, 3], type: "text", value: "123" },
6       { position: [3, 1], type: "separator", value: ":" },
7       { position: [4, 1], type: "whitespace", value: " " },
8       { position: [5, 1], type: "text", value: "4" },
9     ]
10 */
```

Listing 11: Ergebnis des Scannens einer komplexeren Eingabe

Die Komposition des Parsers in die Klassen `SimpleParser` und `TransformingParser` erlaubt die logische Aufteilung der Arbeit. Die grundlegende Verarbeitung der Eingabe, anhand einer KFG, wird vom `SimpleParser` übernommen, während sich der `TransformingParser` um die anschließende Transformation des generierten Ableitungsbaums kümmert. Beide Klassen sind in der Datei `Parser.js` definiert, wobei lediglich der `TransformingParser` unter dem Namen `Parser` exportiert wird. Hierdurch ist die interne Struktur des Parsers für Aufrufer der API verborgen.

Die Ausgabe des Parsers ist ein Ableitungsbaum sowie ein Array von möglichen Warnungen. Der Baum ist durch die Klassen innerhalb der Datei `ParseTree.js` umgesetzt. Der Baum besteht aus den Nonterminalen und Terminalen der KFG. Die beiden Klassen `NonTerminal` und `Terminal` sind Subklassen des abstrakten Datentyps `Node` aus denen die Bäume zusammengefügt werden. Instanzen vom Typ `NonTerminal` sind die Knoten im Baum, während Instanzen vom Typ `Terminal` lediglich als Blätter im Baum vorkommen dürfen. Jeder Knoten besitzt eine Referenz auf seinen Elternknoten und ein Array an Kindern. `NonTerminal` enthält zusätzlich eine Eigenschaft `type`, die den Typ des Nonterminals bestimmt. Die Typen sind alle diejenigen, die auch in der KFG beschrieben wurden. Die Klasse `Terminal` speichert das zugehörige Token in der Eigenschaft `token`. Im folgenden Listing 12 ist eine teilweise Definition der Klassen dargestellt.

```
1   class Node {
2     parent
3     children = []
4     // ...
5     }
6
7   class NonTerminal extends Node {
8     type
9     // ...
10  }
11
12  class Terminal extends Node {
13    token
14    // ...
15  }
16
17  class NonTerminalType {
18    static ROOT = "root"
19    static LINES = "lines"
20    static LINE_OPTIONS = "line_options"
21    static LINE = "line"
22    static LINE_CONTENT = "line_content"
23    static EMPTY_LINE = "empty_line"
24    static METADATA_LINE = "metadata_line"
25    static METADATA = "metadata"
26    static WHITESPACE = "whitespace"
27    static SEPARATOR = "separator"
28    static TAG = "tag"
29    static TAG_CONTENT = "tag_content"
```

```
30      static TEXT = "text"
31      static NEWLINE = "newline"
32  }
```

Listing 12: Ausschnitte aus den Definitionen für `Node`, `NonTerminal`, `Terminal` sowie `NonTerminalType`

`SimpleParser` nimmt nun die Eingabe und Einstellungen (in Form der Klasse `ParserSettings`) entgegen und ruft innerhalb der Methode `parse()` bei Bedarf den Scanner auf, um Token zur Verarbeitung zur Verfügung zu haben. Anhand dieser Token und den Regeln der KFG wird die Eingabe überprüft und der Ableitungsbaum erstellt. Da die Implementierung der beiden Parser ebenfalls lazy ist, greift sie auf die lazy-Methoden im Scanner zurück und generiert den Baum ebenfalls in Schritten. Das Abarbeiten der Token geschieht mit der Hilfe eines eigenen Iterators, dem sogenannten `LookAheadIterator`. Beim Iterieren mittels eines normalen Iterators werden die Elemente konsumiert, so dass ein erneuter Zugriff nicht möglich ist. Dies bereitet Probleme für Produktionen der KFG, in denen der Parser das folgende Token betrachten muss, da dieses Token dann in seiner eigentlichen Iteration nicht mehr zur Verfügung steht. Der `LookAheadIterator` bietet eine Methode `preview()`, welche die Abfrage des nächsten Elements erlaubt, ohne dieses zu konsumieren. Beim Aufruf von `preview()` wird das zurückgegebene Element zusätzlich in einem internen Buffer des Iterators zwischengespeichert, um es bei der nächsten Abfrage durch die Iterator-Methode `next()` erneut bereitzustellen.

Die KFG zur Verarbeitung der Eingabe ist recht einfach gestaltet und orientiert sich an dem möglichen Aufbau von Geheimtexten. Im folgenden Listing 13 ist die KFG dargestellt. Die Nonterminale sind in Großbuchstaben geschrieben. Die Terminale bestehen aus den Typen der Token. Sie werden innerhalb der Produktionen der KFG mit Kleinbuchstaben geschrieben. Als Start-Nonterminal dient `ROOT`.

```
N = { ROOT, LINES, LINE_OPTIONS, EMPTY_LINE, LINE, METADATA_LINE,
  LINE_CONTENT, WHITESPACE, SEPARATOR, TAG, TAG_CONTENT, TEXT, METADATA,
  NEWLINE }
T = { whitespace, newline, separator, opening_tag, closing_tag, metadata
  , text }
S = ROOT

Produktionen:
ROOT          ::= LINES

LINES         ::= LINE_OPTIONS LINES
                | ϵ
LINE_OPTIONS  ::= EMPTY_LINE
                | METADATA_LINE
                | LINE

EMPTY_LINE    ::= WHITESPACE EMPTY_LINE NEWLINE
                | ϵ
METADATA_LINE ::= WHITESPACE METADATA NEWLINE
```

```
LINE          ::= LINE_CONTENT NEWLINE LINE
LINE_CONTENT  ::= WHITESPACE
                | SEPARATOR
                | TAG
                | TEXT

WHITESPACE    ::= whitespace
SEPARATOR     ::= separator

TAG           ::= opening_tag TAG_CONTENT closing_tag
TAG_CONTENT   ::= WHITESPACE
                | NEWLINE
                | TEXT
                | TAG
                | SEPARATOR

TEXT          ::= text
METADATA      ::= metadata
NEWLINE       ::= newline
```

Listing 13: KFG, die der Ableitung des Geheimtextes dient

Die meisten Produktionen aus Listing 13 sind einfach anzuwenden. Zum Beispiel ersetzt die Produktion `WHITESPACE -> whitespace` das `WHITESPACE` Nonterminal immer wenn ein Token vom Typ `WHITESPACE` in der Eingabe angetroffen wurde. Die Produktion `EMPTY_LINE -> WHITESPACE EMPTY_LINE NEWLINE | ` ϵ bedeutet nichts anderes, als dass leere Zeilen aus beliebig vielen Token vom Typ `WHITESPACE` bestehen können und mit einem Zeilenumbruch enden müssen. Lediglich bei der Produktion `LINE_OPTIONS -> EMPTY_LINE | METADATA_LINE | LINE` muss eine aufwendigere Entscheidung getroffen werden. Diese Produktion ersetzt ein Nonterminal vom Typ `LINE_OPTIONS` entweder durch das NonTerminal `EMPTY_LINE`, `METADATA_LINE` oder `LINE`. Wie die Produktionen genau abgearbeitet werden, wird in den folgenden Abschnitten beschrieben.

Während der Abarbeitung verwaltet der Parser neben dem Baum und den Warnungen auch einen Stapel von Nonterminalen. In diesem Stapel ist der initiale Wert, wie auch die Wurzel des Baums, das Nonterminal `ROOT`. Die Methode `parse()` iteriert so lange, bis alle Token aus der Eingabe verarbeitet sind. Nicht jeder Durchlauf der Schleife ruft dabei zwangsweise ein neues Token des Iterators ab. Es kann auch lediglich das Nonterminal im Stapel verändert werden. Sollte innerhalb des aktuellen Schleifendurchlaufs ein Token in dem Baum verändert werden, so wird auch ein neues Token vom Iterator für die nächste Iteration angefordert.

Innerhalb der eigentlichen Schleife der Methode `parse()` wird zunächst das oberste Nonterminal im Stapel betrachtet. Dieses entscheidet, welche Produktion als nächstes

angewandt wird. Während generische Implementierungen eines *Recursive-Decent*-Parser für gewöhnlich anhand von Mengenoperationen (first- und follow-Mengen) entscheiden, welche rechte Seite einer Produktion angewandt werden kann, so nutzt diese Implementierung des Parsers das implizite Wissen, welche Token auf welche Nonterminale passen, um eine einfachere Implementierung zu erhalten. Beim Anwenden von einfachen Produktionen, wie `WHITESPACE -> whitespace` oder `TEXT -> text`, wird lediglich das aktuelle Token als Kind in den Baum eingefügt und das oberste Nonterminal vom Stapel entfernt.

Beim Anwenden von komplexeren Produktionen, wie `EMPTY_LINE -> WHITESPACE EMPTY_LINE NEWLINE | ` ϵ, wird zunächst das aktuelle Token analysiert. Dieses muss zum erwarteten Wert der rechten Seite der Produktion passen. Also ein passendes Token, das durch die Produktion `WHITESPACE -> whitespace` oder `NEWLINE -> newline` vollständig abgearbeitet wird. Hier arbeitet der Parser wieder zur Vereinfachung nicht mittels Mengen, sondern anhand von implizitem Wissen. Sofern ein Token vom Typ `WHITESPACE` angetroffen wird, wird das Nonterminal `WHITESPACE` auf den Stapel gelegt und die Abarbeitung der Schleife beginnt von vorne. Dies geschieht, da bekannt ist, dass das `WHITESPACE` Nonterminal durch ein `WHITESPACE` Token in der Produktion `WHITESPACE -> whitespace` restlos ersetzt werden kann. Beim Antreffen eines Token vom Typ `NEWLINE` ist die Produktion abgearbeitet und das aktuelle Nonterminal (also `EMPTY_LINE`) kann vom Stapel entfernt und das `NEWLINE`-Nonterminal auf den Stapel gelegt werden.

Die Produktion `LINE_OPTIONS -> EMPTY_LINE | METADATA_LINE | LINE` ist die Einzige, die zusätzlich zum aktuellen Token auch das folgende Token der Eingabe betrachten muss. Nur so kann eine korrekte Anwendung der passenden rechten Seite gewährleistet werden. In den vorherigen beschriebenen Produktionen war die Auswahl der passenden rechten Seite anhand des aktuellen Token immer eindeutig. Es musste lediglich überprüft werden, welches Token als Erstes in der nächsten Produktion vorkommt. Dieses Verfahren funktioniert in der aktuell betrachteten Produktion nicht, da die Nonterminale auf der rechten Seite ebenfalls aus komplexeren Produktionen bestehen. Um entscheiden zu können, ob eine `METADATA_LINE` folgt, muss das aktuelle Token vom Typ `METADATA` oder das aktuelle Token vom Typ `WHITESPACE` und das Nachstehende vom Typ `METADATA` sein. In der Implementierung wird diese Entscheidung auch händisch getroffen, da diese einfach genug zu implementieren ist. Für den Fall, dass das nächste Token in der Eingabe leer ist, kann das aktuelle `LINE_OPTIONS`-Nonterminal vom Stapel entfernt werden. In jedem Fall wird das folgende Nonterminal der rechten Seite auf den Stapel gelegt und in den Baum integriert.

Die nächste Iteration muss jeweils das neue oberste Element des Stapels betrachten. Sollte ein unerwartetes Token in der Eingabe erscheinen, so gehört dieses dennoch in den aktuellen Baum integriert, allerdings muss ebenfalls eine zur Situation angepasste Fehlermeldung erzeugt und am Ende der Abarbeitung zurückgegeben werden. Das Einfügen des fehlerhaften Tokens geschieht, um keine Daten der Eingabe auszulassen. Während

der Integration eines Terminals oder Nonterminals in den Baum, wird mittels `yield` der aktuelle Baum und alle bisherigen Warnungen zurückgegeben, um den Aufrufer über den Fortschritt zu informieren.

Abbildung 13 zeigt den mittels *Graphviz*[15] generierten Baum, der beim Parsen der Eingabe `#METADATA value\n1,2\n3 4` entsteht. Die Trennung der Homophone soll in dem Beispiel durch ein Komma realisiert und Leerraum soll beibehalten werden. Die schwarzen Knoten sind die Nonterminale im Baum, während die weiß hinterlegten Blätter die Terminale darstellen. In den Terminalen wird der Inhalt des zugehörigen Token dargestellt. Die Positionen in der Abbildung bestehen wieder aus Index und Länge. Gut zu sehen ist die Strukturierung der verschiedenen Zeilen der Eingabe sowie das Vorkommen der Terminale als Blätter im Baum.

Diese Implementierung des `SimpleParser` genügt, um den eingegebenen Geheimtext passend seiner Struktur in einen Baum umzuwandeln und bestimmte Fehler in der Eingabe zu erkennen. Um komplexere Regeln der Aufteilung von Homophonen in der Eingabe durchzuführen oder bestimmte Features hinsichtlich der Transformation des Geheimtextes einfach zu unterstützen, werden auf dem generierten Baum weitere Transformationen durchgeführt. So sind zum Beispiel das Trennen von Homophonen nach einer bestimmten Anzahl an Zeichen und das Entfernen von Leerzeichen oder Zeilenumbrüchen nicht sinnvoll durch die KFG zu beschreiben. Die nötigen Transformationen sind in der Datei `TreeTransformation.js` implementiert. Viele Transformationen sind dabei essentiell zur Verarbeitung des Geheimtextes. Um dies für den Aufrufer möglichst transparent zu gestalten, werden die Transformationen durch den `TransformingParser` nach dem eigentlichen Parsing des `SimpleParser`s ausgeführt. Neben den Baum-Transformationen werden ebenfalls Konsistenzprüfungen auf dem generierten Baum durchgeführt. Viele dieser Konsistenzprüfungen wären ebenfalls nur umständlich in den Parser zu integrieren. Durch das Ausführen von Transformationen nach dem Parsing ist es zusätzlich nötig, etwaige neu eingeführte Fehler erkennen zu können. Die Konsistenzprüfungen sind in der Datei `TreeConsistencyChecker.js` definiert. Der `TransformingParser` ruft die Methode `parse()` des `SimpleParser`s auf, um den aktuellen Fortschritt des generierten Baums zu erhalten. Auf diesem Baum werden dann die zu den aktuellen Einstellungen passenden Implementierungen zur Transformation und Prüfung ausgeführt.

Im Konstruktor der Klasse `TransformingParser` werden anhand der `ParserSettings` die Baum-Transformationen und die Konsistenzprüfungen initialisiert. Die jeweiligen Transformationen beziehungsweise Konsistenzprüfungen sind durch Klassenhierarchien umgesetzt. Die Oberklassen `TreeTransformation` und `TreeConsistencyChecker` implementieren jeweils das nötige Grundverhalten. Soll zum Beispiel Leerraum aus der Eingabe entfernt werden, so wird die Klasse `WhitespaceRemovalTreeTransformation` als Verarbeitungsschritt im `TransformingParser` integriert. Um die Trennung von Homophonen anhand einer festen Länge zu unterstützen, gibt es die Transformationen

[15] Siehe: `https://www.graphviz.org`

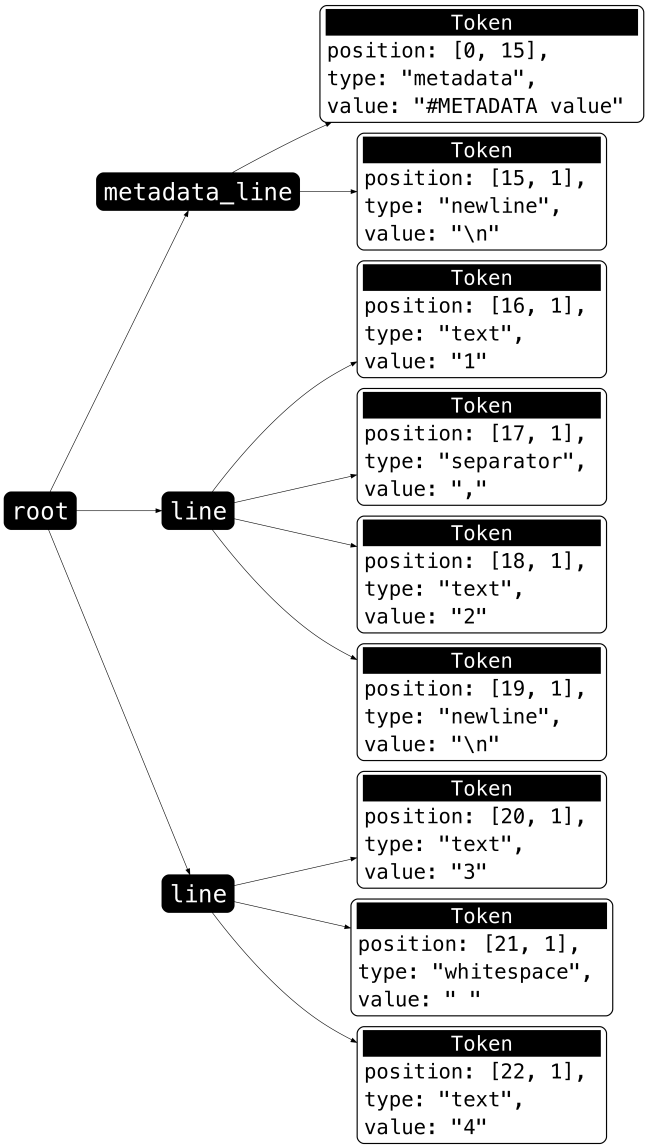

Abbildung 13: Mittels *Graphviz* visualisierter Baum aus der Eingabe `#METADATA value\n1,2\n3 4`

`TextTokenSplittingTreeTransformation` und `TextTokenCombinationTreeTransformation` sowie die Konsistenzprüfung `HomophoneLengthChecker`. Durch die beiden Transformationen ist gewährleistet, dass lange Zeichenketten im Geheimtext, die zusätzlich durch Leerzeichen oder Zeilenumbrüche getrennt sind, dennoch die gewünschte Länge erhalten. Die Konsistenzprüfung kann anschließend etwaige Fehler in der Eingabe erkennen. Fehler können zum Beispiel angemerkt werden, wenn am Ende der Eingabe einzelne Zeichen fehlen oder wenn ein Tag an die falsche Stelle innerhalb der Eingabe geschrieben wurde und damit die erwartete Länge der Homophone verändert.

Die Klasse `TreeTransformation` ist die Oberklasse für die eigentlichen Algorithmen, welche in Unterklassen definiert sind. Die Methode `applyOn()` nimmt einen Baum entgegen und gibt einen neuen, transformierten Baum zurück. Neben den bereits erwähnten Transformationen zum Umsetzten einer festen Länge für Homophone, gibt es unter anderem die nachfolgend beschriebenen Transformationen. So dient `WhitespaceRemovalTreeTransformation` zum Entfernen von Leerraum im Geheimtext. `UnicodeGraphemeClusterJoiningTreeTransformation` hilft bei der einfacheren Umsetzung von Unicode Grapheme-Clustern. Mittels der JS-Bibliothek *graphemer*[16] werden Strings aufgeteilt in Arrays von Unicode Grapheme-Cluster. Vor der Darstellung in der Benutzeroberfläche sollten die Arrays wieder in einen String zusammengeführt werden. Diese Transformation läuft als letzter Schritt in der Verarbeitung, nachdem die Behandlung von Symbolen anhand ihrer Unicode-Darstellung abgearbeitet wurde. Letztlich hilft `DisplayableNodesTreeTransformation` beim Entfernen von Terminalen und Nonterminalen aus dem Baum, deren Darstellung in der Benutzeroberfläche nicht erwünscht ist. Zum Beispiel können mittels dieser Transformation Nonterminale wie `METADATA_LINE` oder Terminale mit Token vom Typ `SEPARATOR` entfernt werden.

Ebenso ist die Klasse `TreeConsistencyChecker` als Oberklasse definiert, deren Methode `check()` von den konkreten Subklassen überschrieben wird. Die Methode bekommt als Parameter ebenfalls einen Baum, gibt allerdings nur ein Array an Warnungen zurück. Konkrete Implementierungen sind unter anderem `DecryptTagNameChecker`, der die Namen der Tags mit den bekannten Tags aus den DECRYPT-Richtlinen vergleicht und bei unbekannten Namen eine Warnung zurückgibt, sowie `UnfinishedTagChecker`, der Tags findet, die in der Eingabe nicht geschlossen wurden. Letztere Konsistenzprüfung ist besonders hilfreich, da der Parser alle Elemente nach einer öffnenden, spitzen Klammer als Text oder Leerraum eines Tags betrachtet.

Die Vorteile der Anwendung von Baum-Transformationen beziehungsweise Konsistenzprüfungen liegen in der einfacheren Implementierung der vielen möglichen Darstellungs-Optionen innerhalb der Benutzeroberfläche, den potentiellen Performance-Optimierungen und der einfachen Integration von Erweiterungen. Dank der Nutzung von Transformationen können die einzelnen Optionen bei Bedarf in die Verarbeitungspipeline eingefügt oder entfernt werden. Hingegen ist zum Beispiel die Trennung der Homophone innerhalb des Scanners oder Parsers aufwendig zu implementieren. Weiterhin kann eine solche feste Implementierung schnell zu Problemen in Kontexten führen, in denen Zeichen in der Eingabe nicht unbedingt zu einem Homophon gehören. Nach dem Öffnen einer spitzen Klammer muss Text als Teil eines Tags und nicht als Homophon erkannt werden. Performance-Optimierungen sind dadurch zu erreichen, dass bei bestimmten Änderungen der Einstellungen der Parser vollständig übersprungen werden kann. So muss zum Aktivieren und Deaktivieren von der Darstellung von Leerraum in der Benutzeroberfläche nicht der gesamte Geheimtext vom Parser erneut eingelesen und verarbeitet werden. Neue Funktionen in der Anwendung können ebenfalls mittels Transformationen, oh-

[16] GitHub-Projekt der Bibliothek graphemer: https://github.com/flmnt/graphemer

ne Änderungen am Parser, integriert werden. Beispielsweise könnten die in Kapitel 3.2 beschriebenen, bisher nicht implementierten Merkmale der DECRYPT Transkriptions-Richtlinien, wie die Umsetzung von Symbolen mit zusätzlichen Markierungen, durch ein Iterieren über die Text-Knoten in einer Baum-Transformation umgesetzt werden. Die Konsistenzprüfungen sind ebenfalls dahingehend anpassbar, so dass die Länge der Homophone auch mit solchen Markierungen korrekt erkennbar ist. Der Text *01⌢.23* könnte mit fester Trennung nach zwei Zeichen zu *01* und *23* verarbeitet werden.

Die Einbindung des Parsers in die Benutzeroberfläche geschieht mittels des React-Hooks `useParser()`, der intern `React.useMemo()` nutzt, um den generierten Baum zwischenzuspeichern. Dieser Hook ruft den Parser mit dem Geheimtext und den Einstellungen auf und liefert den generierten Baum und etwaige Warnungen zurück. Im Fall eines unerwarteten Fehlers, der in der Form einer JS-Exception auftritt, wird dieser in eine reguläre Parser-Warnung umgewandelt und an die Benuteroberfläche übergeben. Der Parser selbst wirft keine Exceptions, so dass dieser Mechanismus nur in wirklich unerwarteten Fällen dazu dient, die weitere Nutzbarkeit der Oberfläche zu gewährleisten.

Um eine einfachere Schnittstelle anzubieten und den Code der Benutzeroberfläche von dem des Parsers zu trennen, gibt es die Klasse `ParseTreeCursor`. Diese besitzt Methoden wie `isEmpty`, `getNumberOfLines()` und `getSymbol(row, column)`, um Anfragen der Benutzeroberfläche entgegenzunehmen. Intern arbeitet der `ParseTreeCursor` auf dem vom Parser generierten Baum, liefert aber anstelle von Token sogenannte `Symbol`s zurück, die unabhängig vom Parser sind und mehr Sinn für die Benutzeroberfläche machen. Zusätzlich ist es mit dieser Abstraktion zukünftig möglich, die lazy-Methoden des Parsers zu nutzen, ohne den Code für die Benutzeroberfläche deutlich zu verkomplizieren.

4.4 Responsive-Design

Um eine sinnvolle Darstellung der Anwendung gleichzeitig auf Desktop- und mobilen Geräten zu ermöglichen, ist die Benutzeroberfläche mittels eines Responsive-Design-Ansatzes umgesetzt. Für viele Komponenten der Oberfläche, wie zum Beispiel dem Geheimtext-Editor oder der Software-Tastatur, ist dies einfach umsetzbar. Die von CTO genutzte Komponenten-Bibliothek *Chakra UI*[17] bietet einige Komponenten, die das automatische Umbrechen von Seitenelementen ermöglichen. So sind unter anderem die Komponenten `Wrap` oder `Grid` zu nennen.

Wie im Kapitel 3.4 beschrieben, ist zusätzlich ein Umbrechen der Zeilen anhand der Breite des Fensters beziehungsweise der Breite des mobilen Geräts erwünscht. Die Umsetzung dieser Anforderung ist anspruchsvoller, da sie nicht nur Auswirkungen auf die Layout-

[17] Homepage von Chakra UI: `https://chakra-ui.com/`

Entscheidungen im Code, sondern auch auf die Implementierung von der Auswahl sowie der Navigation der Symbole in der Hauptansicht hat. Die wichtigste Auswirkung ist die Abbildung von Symbolen auf deren Indizes, abhängig vom aktuellen Kontext. Ist eine Zeile der Eingabe in mehrere sichtbare Zeilen in der Benutzeroberfläche eingeteilt, so ist der Index eines Symbols in einer der umgebrochenen Zeilen nicht mehr identisch mit dem Index in der Eingabe. Soll beispielsweise mittels Pfeiltasten vom ersten Teil der umgebrochenen Zeile in die folgende Zeile navigiert werden, so muss in der Implementierung sicher gestellt sein, dass die Auswahl anhand der Indizes der sichtbaren Zeilen verändert wird. Hingegen ist unter anderem für die Darstellung der eigentlichen Symbole sowie der Nummerierung der Symbolboxen, der Index aus der Eingabe zu nutzen.

Das Listing 14 zeigt ein Beispiel für die unterschiedliche Behandlung von Indizes. Während das ausgewählte Symbol in der Oberfläche den Index `(1, 0)` besitzt, also in der ersten Spalte der zweiten Zeile zu sehen ist, ist der eigentliche Index in der Eingabe `(0, 17)`.

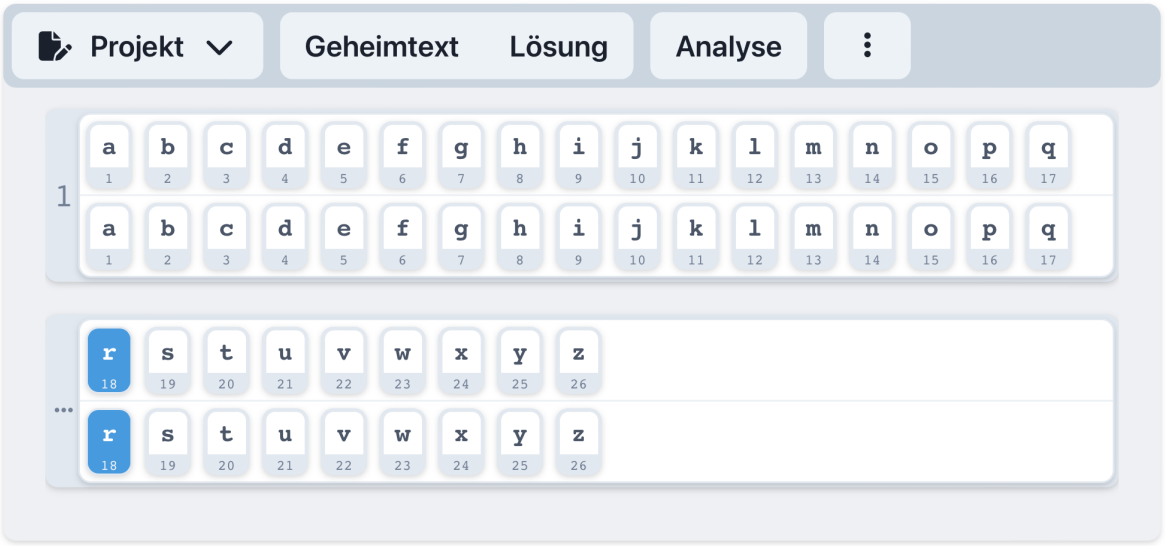

Abbildung 14: Beispiel für unterschiedliche Behandlung von Symbol-Indizes

Neben der direkten Beeinflussung der Auswahl und Navigation, betrifft dieses Verhalten auch alle Aktionen, die auf der Auswahl der Symbole ausgeführt werden sollen. Beispielsweise muss das Einfügen von zusätzlichen Leerzeichen zwischen Symbolen in der Benutzeroberfläche ebenfalls die Indizes der umgebrochenen Zeilen korrekt behandeln.

In einer statisch getypten Sprache wäre die Unterscheidung dieser beiden Arten von Indizes durch unabhängige Typen sinnvoll, so dass eine Nutzung eines bestimmten Indexes an einer falschen Stelle zu einem Compiler-Fehler führen würde. In JavaScript kann mittels Exception ein solches Verhalten emuliert werden, um einen Laufzeitfehler auszulösen, der auf die falsche Nutzung von Indizes opportunistisch während der Entwicklung

aufmerksam macht. In der Implementierung dieser Anwendung wurde bisher auf die Unterscheidung verzichtet, da der Nutzen eines Exception-Mechanismus in diesem Kontext als zu gering erachtet wird.

Auf Grund von einem komplexeren Layout der Zeilen ist das reine Behandeln der Umbrechung mittels Cascading Style Sheets (CSS) nicht möglich. Unter anderem muss bei der Berechnung des Layouts die synchronisierte Breite der Symbolboxen einer Spalte bedacht werden. Diese Elemente sind in unterschiedlichen Container-Elementen in dem HTML-Dokument enthalten, so dass auch aufwendigere Layout-Mittel, wie das Grid-Layout von CSS, eine automatische Umbrechung nicht ermöglichen. Das Berechnen des korrekten Umbrech-Punkts ist also mittels JS-Code durchzuführen. Die `SymbolGrid`-Komponente erfragt seine aktuelle Breite bei jeder Aktualisierung des Fensters oder Rotation des mobilen Geräts. Außerdem besitzt die Komponente eine Hilfsfunktion, die die erforderliche Breite der gesamten Zeile in der Benutzeroberfläche, mittels Angabe des Homophons und dem zugehörigen Klartextsymbol einer jeden Spalte, berechnet.

Um die Logik für die korrekte Behandlung der Auswahl, abhängig von der verfügbaren Breite, übersichtlich und flexibel zu halten, ist das Umbrechen durch einen Decorator um den `ParseTreeCursor`, namens `WrappingParseTreeCursor`, implementiert. Dank des Decorators bleibt die ursprüngliche Schnittstelle des `ParseTreeCursors` erhalten. Das Verhalten der Methoden `getNumberOfLines()` und `getNumberOfSymbolsInLine()`, etc. ändert sich allerdings dahingehend, dass nicht mehr die Werte passend zur Eingabe, sondern zur verfügbaren Breite berechnet werden. Intern nutzt der `WrappingParseTreeCursor` den `ParseTreeCursor` zur Erfragung der Symbole der Eingabe. Das Berechnen der Anzahl der Zeilen und der Länge jeder einzelnen Zeile erfolgt mit den Informationen aus dem `SymbolGrid`, also der insgesamt verfügbaren Breite und der Breite jeder einzelnen Spalte. Ist die Berechnung abgeschlossen, werden die Informationen intern in der Klasse gespeichert und für alle zukünftigen Anfragen auf dieser Instanz genutzt. Ändert sich unter anderem die verfügbare Breite oder die Eingabe des Geheimtextes, so muss das `SymbolGrid` einen neuen `WrappingParseTreeCursor` erzeugen. Neben den vorhandenen Methoden des `ParseTreeCursor` sind zwei weitere Methoden zur Umsetzung der Aktionen der Benutzeroberfläche und der korrekten Behandlung der Indizes der Zeilen und Symbole nötig. Diese bilden zum einen Indizes der Symbole der Darstellung auf die der Eingabe ab und zum anderen die Indizes der Eingabe zurück in die der Darstellung. Da dies nur funktioniert, solange die vorab berechnete Umbrechung der Zeilen zur Darstellung in der Benutzeroberfläche passen, wird ein `WrappingParseTreeCursor` bei jeder Änderung der Darstellung verworfen.

Für etwaige Erweiterungen ist die Umsetzung mittels eines Decorator zusätzlich von Vorteil. Ist zukünftig eine ergänzende Darstellung ohne Umbrechungen erwünscht, so kann eine Instanz des normalen `ParseTreeCursor` genutzt werden, ohne die restliche Implementierung der UI anzupassen.

Während die aktuelle Implementierung die Umbrechung der Zeilen vorab berechnet, wäre diese Berechnung bei Bedarf sinnvoller. Vor allem, wenn das lazy-Verhalten des Parsers vollständig in die Benutzeroberfläche integriert werden soll. Vorteil der aktuellen Implementierung ist, dass die Berechnung meist nur einmalig beim Laden des Dokuments auftritt. Beim Scrollen im Dokument muss daher keine Rechenzeit zum Umbrechen der Eingabe verschwendet werden. Der Nachteil ist, dass eine Veränderung der verfügbaren Breite zu einer kompletten Neuberechnung der Umbrechung für das gesamte Dokument führt, selbst wenn lediglich der Anfang eines (potentiell) großen Dokuments dargestellt wird. Eine weitere Idee zur Vermeidung dieser Laufzeitkosten ist die Einteilung des Geheimtextes in mehrere Seiten. Damit ließe sich der Aufwand zur Berechnung der Umbrechungen auf einen bestimmten Schwellenwert begrenzen.

Zusätzlich zu den bisher erläuterten Erweiterungen des `WrappingParseTreeCursor` ist eine zusätzliche Besonderheit für die Vermeidung von Inkonsistenzen von Relevanz: Ändert sich die verfügbare Breite für die Zeilen oder werden Editierungen vorgenommen, die zu einem Umbrechen der Zeilen führen, während eine Auswahl aktiv ist, so stimmen die bestehenden Indizes dieser Auswahl nicht mehr. Um dieses Problem zu lösen, wird bei jeder Änderung, die zu einem Verwerfen des `WrappingParseTreeCursor` führt, die verworfene Instanz genutzt, um die Indizes der Auswahl zu aktualisieren. Änderungen der verfügbaren Breite, der Abbildungen von Homophonen, der Änderungen an der Eingabe beziehungsweise den Homophonen oder den Darstellungsoptionen, führen zu einem Verwerfen des aktuellen `WrappingParseTreeCursor`. Die Aktualisierung der Auswahl geschieht durch ein Umwandeln der alten dargestellten Indizes in die Indizes der Eingabe mittels der verworfenen Instanz des `WrappingParseTreeCursor` und anschließender Umwandlung dieser Indizes zurück in die der Darstellung anhand der neuen Instanz des `WrappingParseTreeCursor`.

Abbildung 15 verdeutlicht eine Situation, nach der eine Anpassung des aktuellen Indexes einer Auswahl erforderlich ist. Der neue Zustand, im unteren Teil der Abbildung, ist entstanden, indem das letzte Homophon aus der ersten Zeile auf den Text *Beispiel* abgebildet wurde. Auf Grund der dadurch entstandenen Größenänderung der Symbolbox musste diese in die nächste Zeile umgebrochen und der bestehende Index der Auswahl angepasst werden.

4.5 Automatische Analyse

Die automatische Analyse versucht anhand der Verfahren, die in Kapitel 2.2 erklärt wurden, Lösungen für die Abbildung von Homophonen auf Klartextzeichen zu finden. Sie kann die Arbeit bei der Analyse deutlich vereinfachen. Neben der vollständig automatischen Analyse, gibt es auch die Option, solche Algorithmen semi-automatisch zu implementieren. In letzterem Fall ist zusätzlich das Editieren von gefundenen Lösungen möglich, um dem Algorithmus eine Rückmeldung über seine Arbeit zu geben. [14]

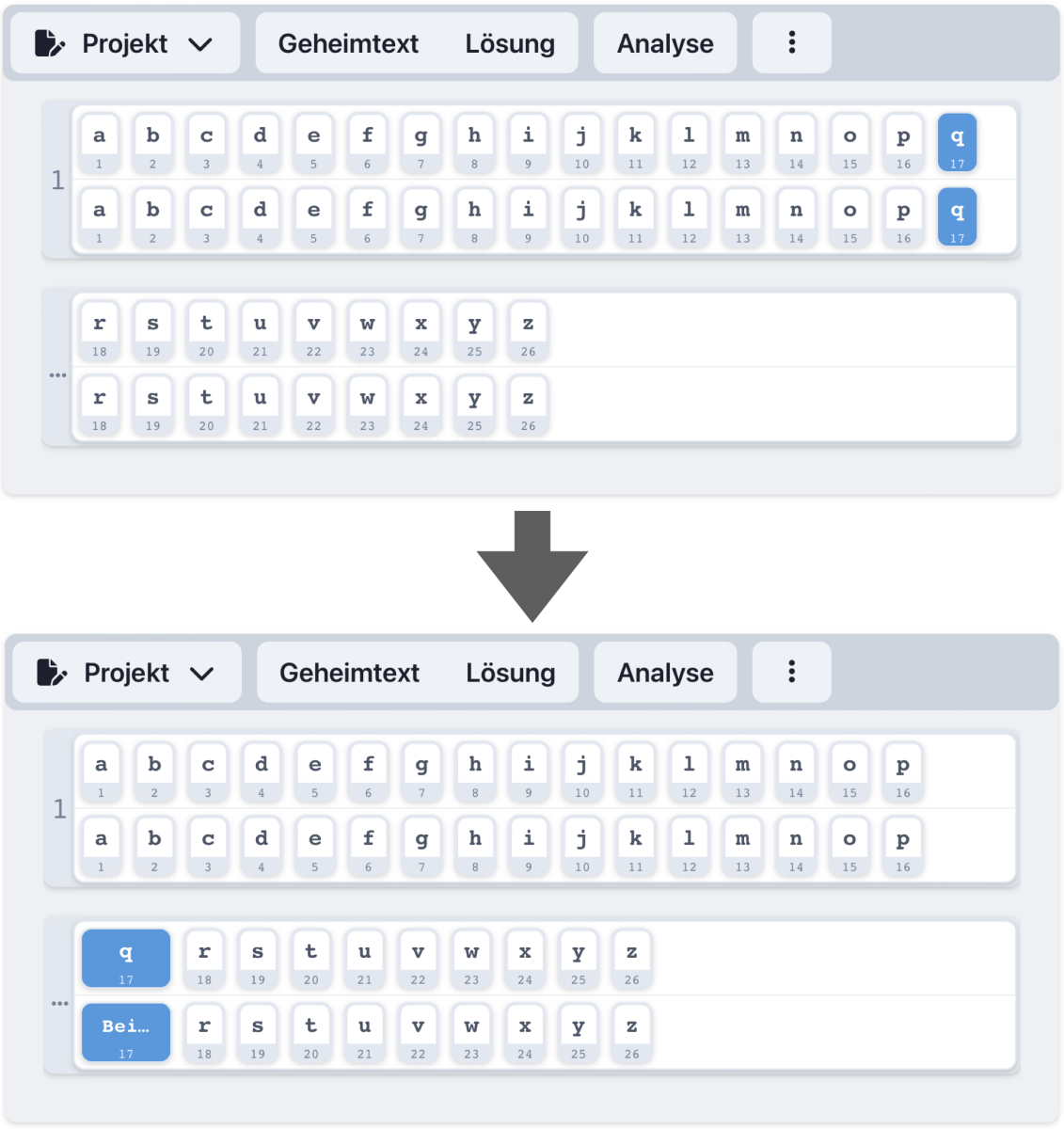

Abbildung 15: Veränderung der Indizes nach einer Editier-Operation

Der aktuelle Stand der Anwendung ermöglicht weder eine voll- noch semi-automatische Analyse von Geheimtexten. Zur späteren Integration ist allerdings eine Schnittstelle definiert, die die einfache Einbindung in die bestehende Benutzeroberfläche ermöglicht. Die Klasse `Analyzer` bildet die Oberklasse für zukünftige Implementierungen von Analyse-Algorithmen. In Listing 14 ist ihre Definition zu sehen. Die Klasse bekommt im Konstruktor ein Array an `Symbol`en, eine Instanz des `HomophoneMapper` und Optionen des Typ `AnalyzerOptions` übergeben. Der `HomophoneMapper` implementiert die Abbildung von Homophonen auf Klartextsymbole, Nomenklatoren sowie Nullen. Die

Methode `perform()` kann aufgerufen werden, um die Analyse zu starten. Dabei kann eine Subklasse des `Analyzer` die Homophone untersuchen und per `HomophoneMapper` unter anderem die aktuellen Klartext-Abbildungen abfragen oder neu eintragen.

```
class Analyzer {
  symbols
  homophoneMapper
  options

  constructor(symbols, homophoneMapper, options) {
    this.symbols = symbols
    this.homophoneMapper = homophoneMapper
    this.options = options
  }

  perform() {}
}
```

Listing 14: Definition der Oberklasse `Analyzer`

CT2 unterstützt den Nutzer bereits bei der Analyse von homophonen Substitutionschiffren durch einen automatischen Analysealgorithmus [14]. Zu Beginn der Arbeit gab es die Überlegung, diesen Algorithmus in die neu zu entwickelnde Anwendung zu integrieren. CT2 ist in *C#* und *.NET* geschrieben und implementiert seine Benutzeroberfläche mittels des *Windows-Presentation-Foundation*-Frameworks [15]. Daher ist die direkte Nutzung in der neuen CTO-Anwendung nicht möglich. Um eine Integration zu erreichen, gibt es allerdings die Möglichkeit mittels WebAssembly (Wasm) den C#-Code auch von JavaScript aufrufbar zu machen. Auch wenn im Laufe der Arbeit erkannt wurde, dass eine Nutzung der bisherigen Implementierung aus CT2 unerwünscht ist, soll im Folgenden dennoch kurz ein Überblick über die mögliche Integration von C# und JS und die etwaigen Schritte zur Nutzung des bisherigen Algorithmus in dieser Umgebung gegeben werden.

Mit .NET Version 7.0 ist es möglich, mit reinen JS-Anwendungen auf C#-Code zuzugreifen. Sind die benötigten Abhängigkeiten installiert, kann im JS-Code das .NET JavaScript-Modul eingebunden werden. Dieses bietet die notwendige Funktionalität, um JS-Module in C# zu integrieren und C#-Code in JS zu nutzen. Das *Marshalling* der Werte und die *Garbage Collection* werden von der .NET-Laufzeitumgebung übernommen. [20]

```
  import { dotnet } from './dotnet.js'

  const { setModuleImports, getAssemblyExports, getConfig } = await dotnet
      .withDiagnosticTracing(false)
      .withApplicationArgumentsFromQuery()
      .create()

  setModuleImports('main.js', {
      example: () => "result",
```

```
10        value: () => 42
11    })
12
13    const config = getConfig();
14    const exports = await getAssemblyExports(config.mainAssemblyName);
15    const result = exports.CSharpClass.Method();
16    console.log(result);
17
18    await dotnet.run()
```

Listing 15: .NET Wasm Interop in JS

Listing 15 zeigt einen minimalen Rahmen um die Interoperabilität zwischen C# und JS herzustellen. Nach dem Import des .NET-Moduls werden die benötigten Funktionen des Moduls abgefragt. Zeile 8 registriert zwei JS-Methoden (`example()` und `value()`) für die Nutzung im C#-Code. `setModuleImports()` ist aus Sicht von C# zu verstehen. Importiert wird der JS-Code und exportiert der C#-Code. Ab Zeile 13 sind die Schritte zum Erfragen und Ausführen einer C#-Methode zu sehen. In Zeile 15 wird so die hypothetische C#-Implementierung der Methode `Method` auf der Klasse `CSharpClass` aufgerufen. Zuletzt wird die .NET-Laufzeitumgebung gestartet und der Einstiegspunkt im C#-Code aufgerufen.

Um die Integration in eine React-App zu erreichen, müssen weitere Schritte vollzogen werden, da das Importieren der `dotnet.js` auf Grund von *Bundlern* wie *Webpack*[18] zu Fehlern führt. Die grundsätzliche Idee ist ein zusätzliches, von React unabhängiges, JS-Skript zu erstellen, dessen Aufgabe darin besteht, die Importe und Exporte mittels der `dotnet.js` umzusetzen.[19] Dieses Skript kann dann mittels dynamischem JS-Import[20] in der Form `await import(/* webpackIgnore: true */ "/script.js")` in die React-App integriert werden.

4.6 Performance-Optimierungen

Ein Ziel der Anwendung ist eine einfache und flüssige Analyse zu ermöglichen. Die Geschwindigkeit der Anwendung, vor allem während der Interaktion, ist dabei von großer Bedeutung. Sowohl die Navigation innerhalb der Darstellung, als auch das Editieren der Symbole soll ohne große Verzögerungen möglich sein. Während der Entwicklung sind immer wieder Situationen aufgetreten, in denen eine Optimierung der Implementierung nötig waren. Zwei elementare Gründe für Einschränkungen der Leistungsfähigkeit sind

[18] Siehe: https://webpack.js.org/

[19] Ein ausführlicheres Beispielprojekt ist in dem folgenden GitHub-Projekt zu finden: https://github.com/maraf/dotnet-wasm-react

[20] Mehr Informationen zu dem dynamischen Import in JS: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import

zu häufige Neuberechnungen von Komponenten sowie große Geheimtexte. Die implementierten Lösungen sollen in den nächsten Abschnitten im Detail vorgestellt werden.

Wie in Kapitel 2.8 beschrieben, wird durch eine Veränderung des Zustands eine Komponente und deren Kinder-Komponenten neu berechnet und dargestellt. In der Anwendung kann dieses Verhalten allerdings an einigen Stellen zu inakzeptablen Verzögerungen führen. Eine dieser Situationen tritt beim Bewegen der Maus über die Symbolboxen auf. Dabei findet eine hellgraue Hervorhebung, der aktuell unter der Maus befindlichen Box, in der Benutzeroberfläche statt. Der Index der hervorzuhebenden Symbolbox wird in der `SymbolRow`, also einer Zeile von Symbolen, gespeichert. Dies dient dazu eine Mehrfachhervorhebung zu vermeiden. Andernfalls kann ein Wechsel der aktiven Anwendung auf einem Desktopsystem dazu führen, dass mehrere `SymbolBox`-Komponenten davon ausgehen, dass der Maus-Cursor über ihrem Bereich schwebt. Das liegt daran, dass der Webbrowser nach dem Anwendungswechsel keine Mausbewegungen mehr erkennt, so dass das Verlassen der Maus aus dem definierten Bereich, der Symbolbox nicht mitgeteilt werden kann. Das Verwalten des Zustands pro Zeile führt allerdings dazu, dass ein Bewegen der Maus durch eine Zeile, beim Erreichen einer anderen Symbolbox, den Zustand der `SymbolRow` ändert und somit eine Neuberechnung aller Symbolboxen in dieser Zeile veranlasst. Je nach Breite des Bildschirms, der Länge der Homophone und der Darstellungsoptionen, können dies üblicherweise zwischen 20 und 50 Symbolboxen sein. Erforderlich ist lediglich das Abändern von maximal zwei Symbolboxen. Nämlich der Symbolbox, die hervorgehoben war und der Symbolbox, die neu hervorgehoben werden soll. Eine andere Situation, die zu einer Neuberechnung von vielen Komponenten führt, ist die Auswahl eines Symbols, zum Beispiel per Mausklick. Die Auswahl wird, wie in Kapitel 4.2 erläutert, im `SymbolGrid` verwaltet. Eine Auswahl einer (oder mehrerer) Symbolboxen führt somit letztlich zu einer Neuberechnung aller Symbolboxen. Bei großen Geheimtexten können schnell hunderte bis tausende Symbolboxen betroffen sein, obwohl diese keinerlei Änderungen erfahren haben.

Vor allem die Auswahl von Symbolboxen in großen Dokumenten führt sowohl im *Debug-Modus* als auch im *Production-Modus* zu einer unzureichenden Leistung der Anwendung. Die Unterschiede der beiden Modi liegen unter anderem in der Anzahl der generierten Warnungen in der JS-Konsole und der Größe der generierten JavaScript-Dateien [18]. Obwohl das Hervorheben von Symbolboxen unter dem Maus-Cursor nur im Debug-Modus merkliche Verzögerungen bewirkt, ist auch hier eine Optimierung sinnvoll, um während der Entwicklung keine unnötigen Leistungseinschränkungen zu erfahren. So ist eine langsame Ausführung während dem manuellen Testen der Implementierung störend. Vor allem können neu eingeführte Geschwindigkeitseinschränkungen eventuell maskiert und somit erst in späteren Tests mit dem Production-Modus erkannt werden. Neben Vorteilen der Optimierung im Hinblick auf die Nutzbarkeit, ergeben sich auch Verbesserungen im Bezug auf den Energiebedarf, insbesondere bei mobilen Geräte. Ein unnötiges Berechnen von unveränderten Komponenten kann auf Dauer die Batterie-Laufzeit des mobilen Geräts einschränken.

Ein Grund für die unzureichende Performance in den besprochenen Situationen ist die Umsetzung von CSS-Stilen mittels CSS-in-JS innerhalb der Bibliothek *Chakra UI*. Diese Umsetzung nutzt die Bibliotheken *emotion* und *styled-system*. Mit CSS-in-JS können CSS Stile direkt in JS beschrieben werden. Das Hinzufügen von CSS-Stilen zu React-Komponenten kann mit CSS-in-JS direkt per `props` geschehen. Beispielsweise kann die *Chakra UI*-Komponente `Box` folgendermaßen mit einem schwarzen Hintergrund eingefärbt werden: `<Box backgroundColor="black" />`. Dieser komfortable Ansatz benötigt allerdings Rechenzeit bei jedem Zeichnen einer Komponente. Die Bibliotheken müssen die CSS-Stile berechnen, die zugehörigen CSS-Klassennamen erzeugen und die Ergebnisse in das DOM serialisieren. Besteht eine Anwendung aus einer Vielzahl von häufig ändernden Komponenten kann dieser Ansatz die Leistungsfähigkeit folglich negativ beeinflussen. [3][8][21]

Abbildung 16 zeigt ein Flammendiagramm aus dem Profiler des Webbrowsers *Firefox*, welches das Navigieren zwischen zwei Symbolboxen visualisiert. Die rot eingerahmte Funktion `withEmotionCache` und deren aufgerufenen Funktionen benötigen bereits ungefähr 30 Prozent der gesamten Rechenzeit dieses Navigationsschritts. Die Implementierung der Anwendung zur Aktualisierung der Zeilen und Symbolboxen benötigt hingegen zusammen deutlich unter 20 Prozent der Rechenzeit.

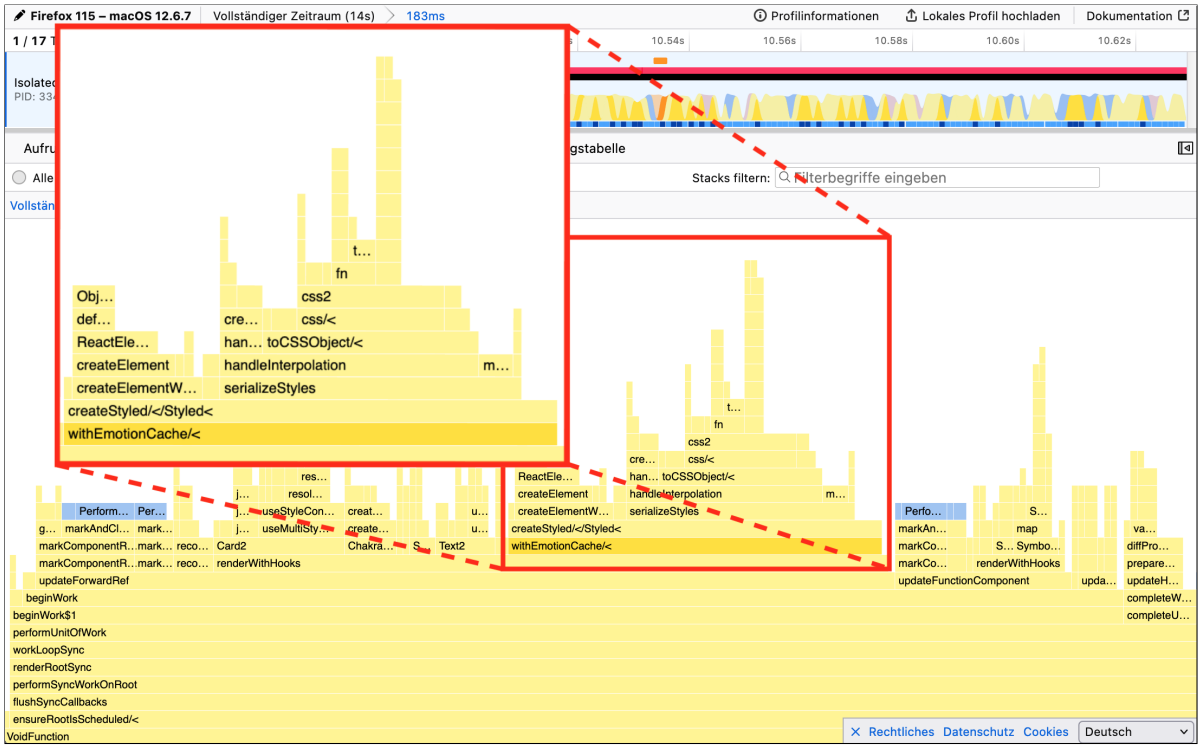

Abbildung 16: Flammendiagramm einer Navigation zwischen zwei Symbolboxen

Um unnötiges, erneutes Berechnen von Komponenten zu verhindern, gibt es die React-Funktion `memo()`[21]. Diese Funktion erhält zwei Parameter und gibt als Ergebnis eine neue Komponente zurück. Der erste Parameter ist die ursprüngliche Komponente, die zwischengespeichert werden soll. Als zweiter Parameter dient optional eine Funktion, welche entscheidet, wann ein erneutes Berechnen der Komponente erforderlich wird. Diese Entscheidung kann die Vergleichsfunktion anhand der alten und neuen `props` der Komponente treffen. Soll die Komponente nicht neu berechnet werden, so gibt die Vergleichsfunktion den Wert `true` zurück. Andernfalls zeigt sie mit dem Rückgabewert `false` an, dass sich Eigenschaften verändert haben. Welche Eigenschaften zum Vergleich genutzt werden, ist der Funktion überlassen. Das Listing 16 zeigt eine gekürzte Definition einer solchen Komponente für `SymbolBox`. Anhand des Vergleichs der wichtigen Eigenschaften der Komponente, wird entschieden, ob eine erneute Berechnung nötig ist. Ändert sich zum Beispiel die von der Komponente `SymbolRow` verwaltete Eigenschaft `isMouseOver`, muss `SymbolBox` neu berechnet werden. Bewegt sich also die Maus über eine, bisher nicht hervorgehobene Box, wird die Vergleichsfunktion `false` zurückgeben, da die Eigenschaft `isMouseOver` unterschiedlich ist. React wird die Funktion der Komponente `SymbolBox` somit erneut ausführen und die neue Version in der Benutzeroberfläche darstellen. Alle anderen Symbolboxen in der Zeile, deren Eigenschaft `isMouseOver` nicht verändert wurde, benötigen keine erneute Rechenzeit.

```
1  const MemoizedSymbolBox = React.memo(SymbolBox, (oldProps, newProps) =>
    {
2    return (
3      oldProps.symbol.isEqual(newProps.symbol) &&
4      oldProps.index.isEqual(newProps.index) &&
5      oldProps.isSelected === newProps.isSelected &&
6      oldProps.isMouseOver === newProps.isMouseOver &&
7      oldProps.isHighlighted === newProps.isHighlighted
8      // ...
9    )
10 })
```

Listing 16: `React.memo()` zur Optimierung von `SymbolBox`

Die Nutzung von `React.memo()` hat allerdings auch Nachteile. Bei jeder Veränderung der `props` einer so definierten Komponente muss ebenfalls die Vergleichsfunktion angepasst werden. Andernfalls kann es zu unerwartetem Verhalten in der Anwendung kommen. Neu hinzugefügte Eigenschaften, die nicht in der Vergleichsfunktion geprüft werden, beeinflussen nicht den Rückgabewert dieser Funktion, können also nicht das Neuberechnen der Komponente und somit eine Veränderung in der Benutzeroberfläche herbeirufen. Generell ist es laut [17] nicht sinnvoll jede Komponente per `React.memo()` zu definieren, da dies den Code unleserlicher macht und nur bei bestimmten Komponenten zu einer Verbesserung der Geschwindigkeit führt. Komponenten, die von `React.memo()` profitieren sind demnach besonders feinteilig und ändern sich nicht bei jeder Aktualisierung der Benutzeroberfläche mit.

[21] Dokumentation zur Funktion `memo`: https://react.dev/reference/react/memo

Dieser Abschnitt soll die Geschwindigkeitsvorteile von `React.memo()` vorstellen. Zur Messung der Geschwindigkeit wurde der `React.Profiler`[22] genutzt. Diese Komponente bietet eine Callback-Funktion, welche unter anderem die gemessene Zeit für das Zeichnen aller Kind-Komponenten des `React.Profiler` bereitstellt. Die Untersuchung geschieht dabei im Debug-Modus. Die Werte entsprechen somit nicht unbedingt denen, die in der fertigen Anwendung auftreten werden, zeigen aber dennoch den Unterschied zwischen den Implementierungen. Um belastbare Werte zu erhalten, wurden alle zu messenden Aktionen 1000-fach wiederholt und der Median ermittelt. Zur Messung der Geschwindigkeitsunterschiede zwischen dem Zeichnen der Anwendung mit und ohne `React.memo()`, dienen die beiden, am Anfang des Kapitels beschriebenen, optimierungsbedürftigen Situationen. Bei diesen Situationen handelt es sich um das Hervorheben von Symbolboxen bei Bewegung der Maus und das Navigieren innerhalb der Anwendung. Obwohl beide Messungen nicht vollständig isoliert stattfinden und von dem Nutzerverhalten abhängen, so sollen die Ergebnisse dennoch einen Eindruck der Geschwindigkeitsvorteile vermitteln. Bei den Messungen kommt derselbe Geheimtext mit denselben Einstellungen zum Einsatz. Dieser Geheimtext führt zu einer Darstellung von 59 Zeilen und insgesamt 2.750 Symbolboxen. Einziger Unterschied ist die Nutzung von `React.memo()` im Code. Zunächst soll das Hervorheben der Symbolboxen betrachtet werden. Dazu fährt die Maus wiederholt über alle Symbolboxen von zwei Zeilen der Hauptansicht. Ohne die Nutzung von `React.memo()` beträgt der Median zur Zeichnung der Komponente 86,7 ms während er mit Nutzung von `React.memo()` lediglich 15,6 ms beträgt. Der Test der Navigation erfolgt nach einem ähnlichen Prinzip. Mittels Pfeiltasten soll ebenfalls über alle Symbole von zwei Zeilen der Darstellung navigiert werden. Hierbei sind mehr Komponenten involviert, als bei dem vorherigen Test, wodurch sich die längeren Zeiten erklären. Der Unterschied in der Leistungsfähigkeit der Anwendung ist noch deutlicher. Ohne `React.memo()` beträgt der Median einer jeden Zeichnung der Anwendung 1019,6 ms und mit `React.memo()` lediglich 139,7 ms. Diese Unterschiede sind auch in der normalen Nutzung der Anwendung deutlich zu merken. Vor allem ein flüssiges Editieren des Geheimtextes ist bei einer Verzögerung von ungefähr einer Sekunde pro Navigationsschritt unmöglich. Dieser Wert verbessert sich auch nicht wesentlich im Production-Modus der Anwendung. Die gemessenen Zeiten verdeutlichen, dass eine Nutzung von `React.memo()`, trotz der erwähnten Nachteile, in der Anwendung unumgänglich ist.

Neben den Geschwindigkeitseinbußen bei häufigem Neuberechnen können ebenfalls große Geheimtexte zu schlechter Performance führen. Bei fester Länge der Homophone steigt die Anzahl der dargestellten Symbolboxen proportional zur Länge des Geheimtextes. Jede Symbolbox besteht aus sieben HTML-Elementen im generierten HTML-Dokument. Neben dem eigentlichen Homophon oder Klartext-Symbol, wird in einer Symbolbox auch ein Index angezeigt sowie mittels Container-Elementen ein ansprechendes Layout erreicht. Die *Google Lighthouse*-Dokumentation[23] warnt vor zu großen

[22] Dokumentation des `React.Profiler`: https://react.dev/reference/react/Profiler

[23] Google Lighthouse-Dokumentation: https://developer.chrome.com/docs/lighthouse/performance/dom-size/

HTML-Dokumenten, da diese unter anderem die Laufzeit- und Speicher-Performance einschränken. Um große Geheimtexte zu unterstützten muss also die Anzahl an generierten HTML-Elementen eingeschränkt werden. Eine Möglichkeit ist ein *Paging*, bei dem der Geheimtext in mehrere virtuelle Seiten eingeteilt wird. Allerdings leidet die Übersicht unter einer solchen Einteilung. Außerdem müssten diese virtuellen Seiten, wie später noch beschrieben wird, dennoch ziemlich klein gehalten werden, um eine brauchbare Leistungsfähigkeit zu erhalten. Alternativ kann die Größe des DOM durch ein *Windowing* beschränkt werden. Die React-Dokumentation zur Performance-Optimierung spricht sich ebenfalls für ein Windowing als Lösung für größere Datenmengen aus [18]. Besteht die Datenmenge aus tausenden von Zeilen und bietet die Benutzeroberfläche nur Platz für einige dutzend Zeilen, so sorgt das Windowing dafür, dass nur die sichtbaren Zeilen auch tatsächlich im HTML-Dokument existieren. Die benötigte Speichermenge orientiert sich somit an der Größe der verfügbaren Oberfläche anstelle der Datenmenge.

In der Anwendung wird das Windowing mittels der JS-Bibliothek *react-virtualized*[24] umgesetzt. Neben der beschriebenen Funktionalität war es wichtig, dass das Windowing transparent in die Benutzeroberfläche integriert werden kann. So soll es keinen visuellen Unterschied geben, ob das Windowing in der Oberfläche aktiviert ist oder nicht. Die Bibliothek bietet mehrere Komponenten für die Umsetzung des Windowing. Neben der elementaren Komponente `List` gibt es noch weitere Komponenten zum Anpassen des Verhaltens. Für die Anwendung sind die Komponenten `AutoSizer`, `WindowScroller` und `InfiniteLoader` relevant. `List` implementiert das eigentliche Windowing. Innerhalb eines Rahmens, definiert durch die Position sowie die Breite und Höhe, werden die Zeilen gezeichnet. Dabei muss für die Berechnung des Scrollbalkens die Anzahl der Zeilen und deren Höhe per `props` angegeben werden. Letztlich dient die Eigenschaft `rowRenderer` dazu eine Funktion zu definieren, die vor der Darstellung der Zeile aufgerufen wird und eine entsprechende Komponente zur Integration in die Liste zurückgibt. Mittels `AutoSizer` kann die Liste automatisch an die verfügbare Breite und Höhe in ihrem Container angepasst werden. `WindowScroller` ist besonders wichtig, um die transparente Umsetzung des Windowing in der Anwendung zu erreichen. Die Komponente ermöglicht das Scrollen mittels des Scrollbalkens des Fensters. Ohne diese Komponente würde die Liste ihren eigenen Scrollbalken anzeigen. In der Anwendung müssten damit zwei Scrollbalken gezeigt werden. Einer für die eigentliche Seite und ein weiterer für die Liste. Außerdem führt ein Scrollen, je nach Mausposition, dazu, dass sich die Liste oder das äußere Dokument bewegt. Zuletzt ist der `InfiniteLoader` eine Komponente, die für zukünftige Optimierungen in der Anwendung genutzt werden kann. Sie ermöglicht das Laden von Zeilen bei Bedarf. [2]

```
1     <InfiniteLoader
2       isRowLoaded={isRowLoaded}
3       loadMoreRows={loadMoreRows}
4       rowCount={parseTreeCursor.getNumberOfLines()}
5     >
```

[24] GitHub-Projekt der Bibliothek react-virtualized: https://github.com/bvaughn/react-virtualized

```
6       {({ onRowsRendered, registerChild }) => (
7         <WindowScroller>
8           {({ height, isScrolling, onChildScroll, scrollTop }) => (
9             <AutoSizer disableHeight>
10              {({ width }) => (
11                <List
12                  ref={(element) => {
13                    initListRef(element)
14                    registerChild(element)
15                  }}
16                  autoHeight
17                  width={width}
18                  height={height}
19                  isScrolling={isScrolling}
20                  scrollTop={scrollTop}
21                  onScroll={onChildScroll}
22                  onRowsRendered={onRowsRendered}
23                  rowCount={parseTreeCursor.getNumberOfLines()}
24                  overscanRowCount={5}
25                  rowHeight={estimatedRowHeight}
26                  rowRenderer={wrappedRowRenderer}
27                />
28              )}
29            </AutoSizer>
30          )}
31        </WindowScroller>
32      )}
33    </InfiniteLoader>
```

Listing 17: Umsetzung des Windowing mittels *react-virtualized*

Das Listing 17 zeigt die erwähnten Komponenten der Bibliothek *react-virtualized*, wie sie in der Anwendung eingebunden sind. Die Komponenten sind ineinander geschachtelt und bieten ihre jeweilige Funktionalität der `List`-Komponente an. Beispielsweise ist in Zeile 10 erkenntlich, wie `AutoSizer`, der lediglich die Breite vorgeben soll, diese per Funktion mit einem Objektparameter an die `List`-Komponente übergibt. In Listing 18 sieht man einen kleinen Ausschnitt aus der Implementierung der Funktion `wrappedRowRenderer`. Die Parameter `isScrolling` und `isVisible` erlauben weitere Optimierungen, wie zum Beispiel die Darstellung eines Platzhalters, sofern die Liste noch am Scrollen ist.

Bevor das Windowing in der Anwendung eingeführt wurde, war das Umbrechen der Zeilen innerhalb der `SymbolRow` implementiert. Neben der semantischen Ungenauigkeit, funktioniert dieses Vorgehen mit der Einführung der `List`-Komponente nicht mehr. Die Liste benötigt die Anzahl der Zeilen und deren Höhe im voraus. Während diese Komplikation durch den `InfiniteLoader` umgangen werden kann, ist, wie in Kapitel 4.4 beschrieben, die robusteste Variante, die Implementierung der Umbrechung im `WrappingParseTreeCursor`. Dessen Informationen zur Anzahl und Inhalt der Zeilen werden in der Funktion `wrappedRowRenderer` genutzt, um die `SymbolRow`s zu initiali-

sieren. Jede `SymbolRow` implementiert damit auch eine sichtbare Zeile in der Benutzeroberfläche.

```
function wrappedRowRenderer({ key, index, isScrolling, isVisible, style
  }) {

  // ...

  return (
    <div key={key} style={style}>
      <SymbolRow
        rowIndex={rowIndex}
        ciphertextSymbols={ciphertextSymbols}
        plaintextSymbols={plaintextSymbols}
        // ...
      />
    </div>
  )
}
```

Listing 18: Ausschnitt der `wrappedRowRenderer`-Funktion

Zur Evaluation der Vorteile des Windowing soll eine andere Herangehensweise, als bei der Messung des Einflusses von `React.memo()` genutzt werden. Zur besseren Beurteilung des Windowing auf den Speicherbedarf soll ein größerer Geheimtext genutzt werden. Dieser erzeugt 354 Zeilen und insgesamt 8250 Symbolboxen. Ein Vergleich der Webbrowser-Prozesse in der macOS Aktivitätsanzeige verdeutlicht den Speicherbedarf. Der Prozess für den *Safari*-Webinhalt der Anwendung belegt ohne Windowing 3,14 GB und mit Windowing lediglich 625 MB. Neben dem deutlich erhöhten Speicherverbrauch, ist auch die Stabilität des Safari-Webbrowsers eingeschränkt. Beispielsweise sind normale Operationen in der Menüleiste deutlich verzögert. Wird Firefox als Webbrowser genutzt sind die Werte geringer: Ohne Windowing werden 1,27 GB Speicher für den Webprozess verbraucht, mit Windowing 326 MB. Außerdem sind im Firefox die Instabilitäten nicht zu beobachten. Beim Windowing orientiert sich der Speicherbedarf an der Anzahl der darstellbaren Elemente, während ohne Windowing der Speicherbedarf mit der Größe des Dokuments steigt. Neben dem Speicherverbrauch und etwaigen Stabilitätsproblemen des Webbrowsers, können ebenfalls Geschwindigkeitsunterschiede zwischen den Implementierungen festgestellt werden. Die folgenden Messungen haben unter der Nutzung von `React.memo()` stattgefunden, so dass die Implementierung die bestmögliche Leistungsfähigkeit bietet. Zunächst tritt nach der Bestätigung des eingegebenen Geheimtextes eine große Verzögerung auf. Bis die Zeilen und Symbolboxen erstellt sind, blockiert die Anwendung. Ohne Windowing dauert es 23,6 s bis der Geheimtexteditor geschlossen und die Hauptansicht dargestellt wird, mit Windowing 669 ms. Zusätzlich dauern Editieroperationen deutlich länger, da zum Beispiel beim Navigieren von einer Symbolbox zur nächsten, alle Hervorhebungen des aktuell ausgewählten Homophons berechnet werden müssen. Bei großen Dokumenten ist die Vielzahl dieser Hervorhebungen allerdings nicht sichtbar. Das Navigieren zwischen Symbolen in demselben, großen Ge-

heimtext dauert im Median ohne Windowing 3.446 ms und mit lediglich 172,6 ms. Das Beispiel verdeutlicht, dass unabhängig von der Browserstabilität ein Windowing zwangsweise für größere Dokumente implementiert sein muss. Nachteilig an dieser Technik ist hingegen der zusätzliche Aufwand beim Scrollen. Bei kleineren Dokumenten kann das Scrollen ohne Windowing schneller sein. Allerdings verdeutlichen die obigen Messwerte, dass dieser kleine Vorteil nicht schwer genug wiegt, um die enormen Verzögerungen ohne Windowing aufzuwiegen.

Eine zukünftige Optimierung ist die Nutzung des `InfiniteLoader` im Zusammenhang mit der lazy-Implementierung des Parsers. Bisher wird der `List`-Komponente die Anzahl der anzuzeigenden Zeilen vorab übermittelt. Damit müssen sowohl der Parser als auch der Scanner die gesamte Eingabe verarbeiten. Mittels `InfiniteLoader` kann die Verarbeitung der Zeilen des Geheimtextes soweit eingeschränkt werden, dass zunächst lediglich der aktuelle Ausschnitt der Benutzeroberfläche gefüllt ist. Alle weiteren Zeilen können anschließend bei Bedarf über die Eigenschaft `loadMoreRows` des `InfiniteLoader` angefragt werden. Durch dieses Vorgehen kann zum einen der initiale Aufwand der Textanalyse reduziert und zum anderen das Umbrechen der Zeilen für unsichtbare Zeilen vermieden werden. Des Weiteren besteht eine zusätzliche Möglichkeit zur Optimierung bei der Eingabe und dem Editieren von großen Geheimtexten. Derzeit wird jede Änderung im `CiphertextEditor` direkt an den Parser weitergereicht. Dieser baut dann einen neuen Baum auf und zeigt etwaige Warnungen an. Soll der Editor nicht nur zum Einfügen und zur Referenz des Geheimtextes dienen, sondern auch aktiv an einem Geheimtext gearbeitet werden, so kann die direkte Einbindung des Parsers zu störenden Verzögerungen beim Editieren führen. Zunächst ist die Implementierung eines *Debounce*-Mechanismus sinnvoll, um nicht nach jeder Eingabe der Tastatur, den Parser aufzurufen. Ein solcher Mechanismus verzögert die Ausführung des Parsers um einige hundert Millisekunden und bricht eine geplante Ausführung ab, sofern innerhalb des Warteintervalls eine weitere Eingabe im Editor erfolgt. Damit wird der Parser erst ausgeführt, nachdem Änderungen am Geheimtext für einige Zeit Bestand haben. Sollten regelmäßig sehr große Geheimtexte im `CiphertextEditor` editiert werden, ist als weitere Optimierung ein asynchroner Aufruf des Parser denkbar. Wird das Editieren eines Geheimtextes temporär abgeschlossen, so kann die Geheimtext-Analyse im Hintergrund durchgeführt werden ohne die Benutzeroberfläche zu blockieren. Beginnen weitere Editier-Operationen nach dem Warteinterval des Debouncers, werden diese nicht durch die gleichzeitige Verarbeitung des Geheimtextes blockiert.

5 Evaluation

Um während der Entwicklung sicherzustellen, dass die Anwendung den Ansprüchen genügt, wurden regelmäßige virtuelle Besprechungen gehalten. So konnten Verbesserungen und Änderungswünsche schnell eingearbeitet werden. Die Vorstellung des Fortschritts erfolgte per Bildschirmfreigabe. Daneben wurde regelmäßig eine Testversion der Anwendung zur Verfügung gestellt. Die Rückmeldung zu diesen Testversionen war grundsätzlich positiv.

Zur weitergehenden Beurteilung der Anwendung, wurde die Analyse eines unbekannten Geheimtextes angestrebt. In Abbildung 17 ist die erste Seite des originalen, untersuchten Dokuments aus [22] zu sehen. Bekannt war vor der Analyse lediglich, dass der Inhalt des Dokuments in englischer Sprache verfasst ist, dass die einzelnen Zeichen des Geheimtextes mittels eines Punkts getrennt sind und dass der Bindestrich die Verschlüsselung eines Leerzeichen ist. Abbildung 18 zeigt die entschlüsselte Nachricht in der Anwendung. Markierte Wörter sind grün umrandet, die zugehörigen Symbole an den anderen Stellen im Geheimtext grün hinterlegt. Nullen sind mit einer Raute und roten Hintergrund hervorgehoben. Um die Entschlüsselung zu erreichen, wurde zunächst in der Anwendung nach häufig vorkommenden Zeichen und etwaigen Mustern im Geheimtext gesucht. Mit diesem Vorgehen konnte das Wort „the“ als Entschlüsselung der Symbolkette „38 16 10“ angenommen werden. Daneben gibt es ein wiederholtes Muster in den ersten Zeilen des Dokuments, welches mit der vermuteten Entschlüsselung des Zeichen „16“ nach „H“ das Wort „which“ wahrscheinlich macht. In den Zeilen 6 und 8 aus Abbildung 18 sind die markierten Vorkommen des Worts ersichtlich. Auffällig ist außerdem die rote Markierung des Zeichen „13“ als Null hinter dem ersten Vorkommen des Worts. Durch Fortführung dieses Verfahrens konnten immer mehr Wörter vermutet werden. Innerhalb der Anwendung wurden wahrscheinlich korrekte Entschlüsselungen als Wort markiert, wodurch zusätzlich eine Markierung aller in dem Wort vorkommenden Symbole stattfindet. So konnten bestehende Lücken in größtenteils entschlüsselten Wörtern einfacher erkannt und letztlich die gesamte Analyse erfolgreich abgeschlossen werden.

Zur Evaluation der fertiggestellten Anwendung sollte diese zusätzlich von zwei Kommilitonen getestet werden. Ihnen wurde dafür ein einfacher, monoalphabetischer Geheimtext zur Verfügung gestellt, den sie nach einer kurzen Einführung in die Thematik analysieren sollten. Die Einführung diente dazu die Kommilitonen sowohl mit der monoalphabetischen als auch der homophonen Verschlüsselung vertraut zu machen. Des Weiteren sind die Ziele der Anwendung vorab dargestellt worden. Davon abgesehen sollte die Bearbeitung innerhalb der Anwendung ohne größere Erklärungen oder Hilfen geschehen. Es war nicht das Ziel, dass der gegebene Geheimtext entschlüsselt wird, sondern dass die Erfahrungen von Nutzern in die Anwendung einfließen sowie etwaige Ungereimtheiten oder Fehler gefunden werden konnten.

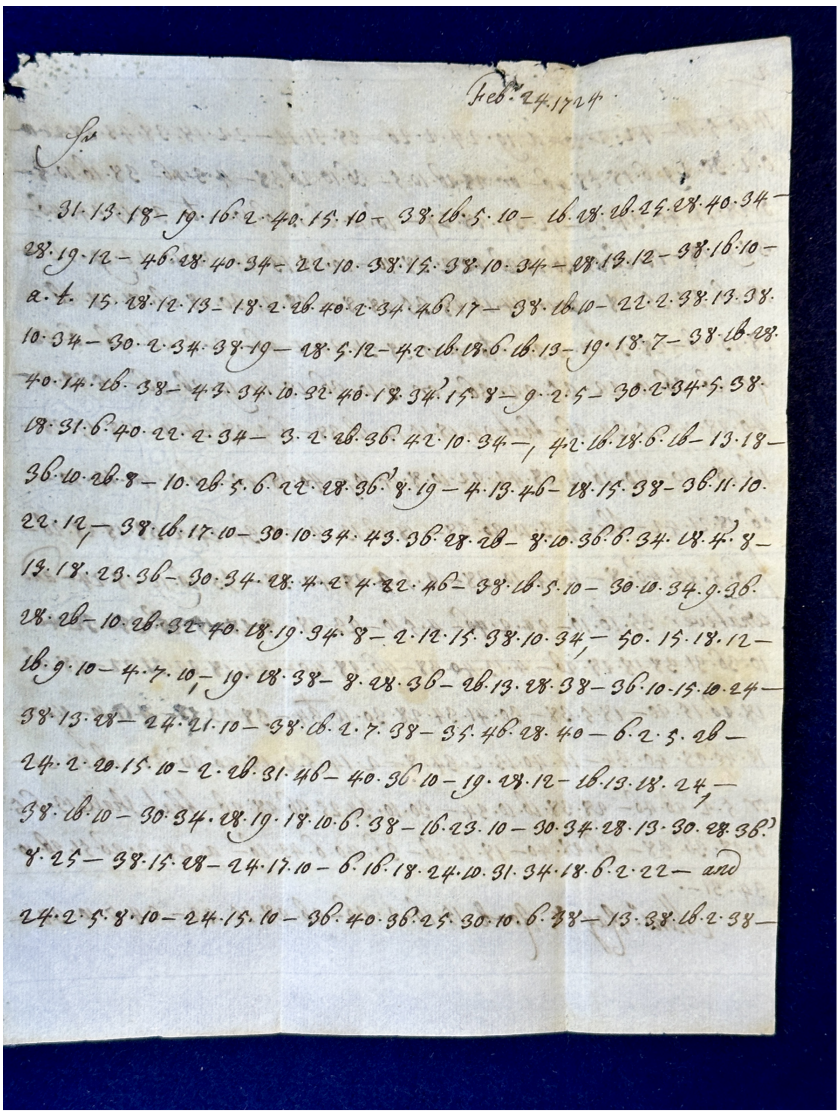

Abbildung 17: Erste Seite der verschlüsselten Nachricht aus [22]

Grundsätzlich war die Evaluation der Anwendungen durch die Kommilitonen erfolgreich. Der gegebene monoalphabetische Geheimtext konnte nach einiger Zeit erfolgreich entschlüsselt werden. Auch die Rückmeldungen waren größtenteils positiv. Während der Nutzung sind allerdings auch einige verbesserungswürdige Details aufgefallen. So hatten beide Kommilitonen zunächst Schwierigkeiten bei der genauen Konfiguration des Geheimtext-Editors. Die Eingabe des gegebenen Geheimtextes gelang ohne Probleme, allerdings waren die vorhandenen Optionen zunächst verwirrend. Um dieser Verwirrung entgegenzuwirken sind, neben einer allgemeinen Benutzungsanleitung, zusätzliche Hilfestellungen im Geheimtext-Editor sinnvoll. Diese Hilfestellungen könnten in Form eines aufrufbaren Hilfetextes direkt im Editor oder als Tooltips an den verschiedenen Optionen implementiert werden.

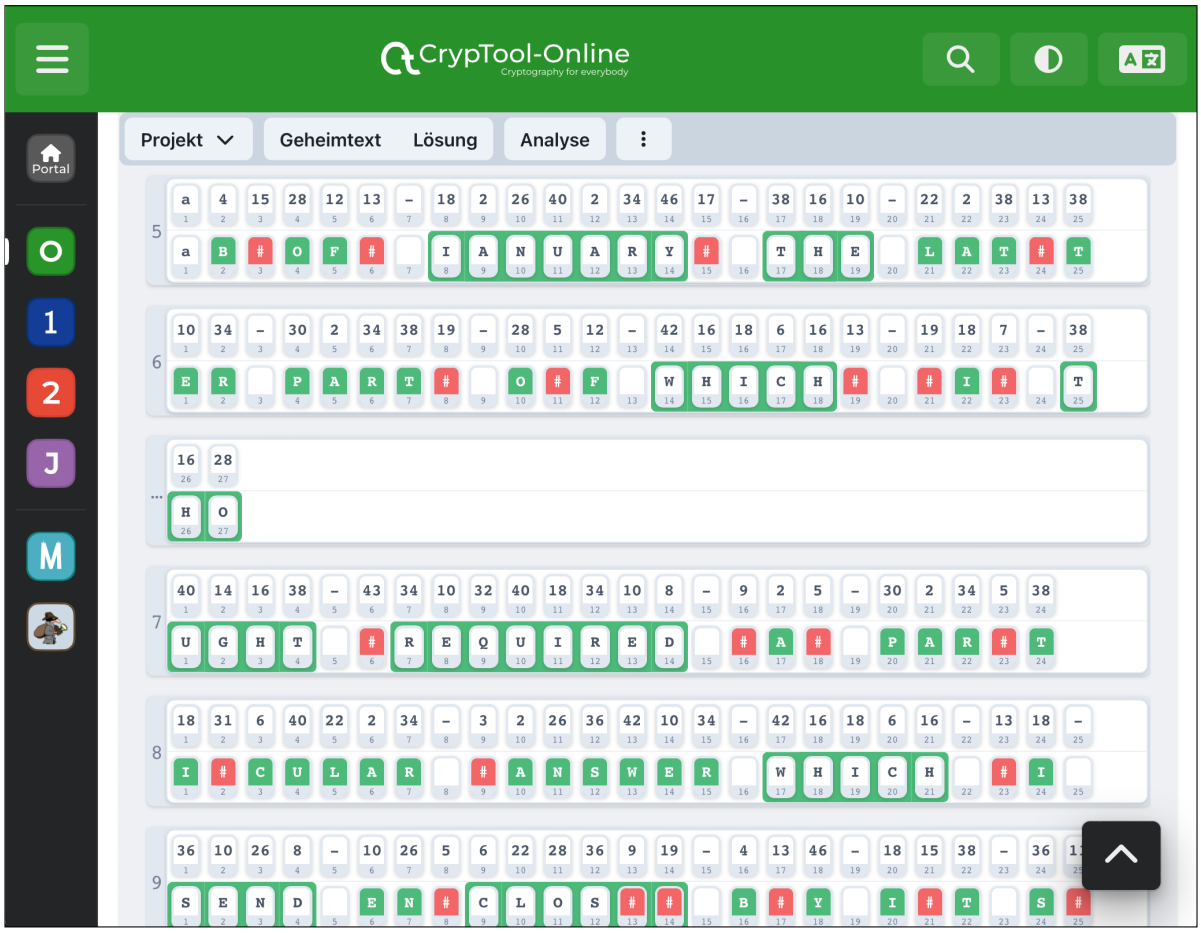

Abbildung 18: Entschlüsselung der analysierten, verschlüsselten Nachricht aus [22]

Besonders gelobt wurden bei diesen Tests die Möglichkeit der Navigation per Tastatur. So war ein angenehmes und flüssiges Bearbeiten innerhalb der Anwendung möglich. Daneben gab es auch einige Ideen für Erweiterungen, die das Analysieren angenehmer gestalten würden. Ein großer Wunsch war die Umsetzung von *Undo- / Redo*-Funktionalität. Auch wenn die angewandten Operationen von den Kommilitonen immer manuell rückgängig gemacht werden konnten, sorgt die Möglichkeit Änderungen direkt über eine Tastenkombination widerrufen zu können für ein sicheres Gefühl. Eine weitere Idee war eine Such- beziehungsweise Filterfunktion, so dass nach Vorkommen von einzelnen Homophonen oder Klartextsymbolen gesucht oder gar Symbole anhand bestimmter Merkmale hervorgehoben werden können.

6 Verwandte Arbeiten

Dieses Kapitel beschreibt andere Anwendungen die zur Kryptonalyse von Substitutionchiffren eingesetzt werden können. Es wurden einige Desktop- als auch Webanwendungen untersucht und letztlich auch mit der entwickelten Anwendung verglichen.

Zunächst ist der *homophone Substitutionsanalysator* in CT2 zu nennen, dessen Adaption für CTO in der entwickelten Anwendung umgesetzt wurde. Die Implementierung in CT2 ermöglicht die manuelle und automatische Analyse von homophonen Substitutionschiffren. In Kapitel 3.4 ist eine kurze Diskussion der Benutzeroberfläche von dem homophonen Substitutionsanalysator in CT2 zu finden.

Es gibt darüber hinaus einige Desktop-Applikationen zur automatischen Analyse von Geheimtexten. Unter anderem sind als Anwendungen mit Unterstützung für homophone Substitutionschiffren *zkdecrypto*[25] und *AZdecrypt*[26] zu nennen. zkdecrypto wurde ursprünglich entwickelt, um die 340-Zeichen lange Nachricht des Zodiac-Mörders zu entschlüsseln. AZdecrypt, von Jarl Van Eycke, unterstützt die Analyse von klassischen Chiffren mittels Hill-Climbing-Algorithmus. Beide Anwendungen erlauben es einen Geheimtext zu laden und unter Angabe etlicher Optionen eine automatische Analyse durchzuführen. AZdecrypt bietet eine geteilte Ansicht, die sowohl Geheimtext als auch Klartext gleichzeitig darstellen kann. zkdecrypto verändert die Darstellung des Geheimtextes während der automatischen Entschlüsselung. Es gibt in beiden Anwendungen allerdings keine Möglichkeit zur manuellen Editierung des entschlüsselten Klartextes. Daher ist weder das Korrigieren von einzelnen Fehlern noch die Beeinflussung der verwendeten Algorithmen möglich. Die Darstellung des Geheimtextes und Klartextes ist wenig übersichtlich. In AZdecrypt wird der Geheimtext entsprechend der Eingabe dargestellt, während zkdecrypto zusätzlich die Vorkommen von einzelnen Homophonen hervorheben kann. Die Trennung der Homophone und die Zugehörigkeit zwischen Homophonen und entschlüsselten Klartextzeichen ist allerdings nicht ersichtlich.

Neben den Desktop-Applikationen gibt es einige Webseiten, die den Umgang mit homophonen Substitutionschiffren beherrschen. Die Seite *CryptoPrograms*[27] bietet die Möglichkeit klassische Chiffren zu erstellen und zu analysieren. Für homophone Substitutionschiffren bietet die Webseite allerdings nur eine Anwendung zum Verschlüsseln von Klartexten an.[28] Diese Anwendung erlaubt die Eingabe von Klartext und kann unter Angabe einiger Optionen einen Geheimtext generieren. Hingegen erlaubt CryptoPrograms unter anderem die automatische Entschlüsselung von monoalphabetischen Substituti-

[25] Google Code-Archiv von zkdecrypto: https://code.google.com/archive/p/zkdecrypto/
[26] GitHub-Projekt von AZdecrypt: https://github.com/doranchak/azdecrypt
[27] CryptoPrograms-Homepage: https://www.cryptoprograms.com/
[28] CryptoPrograms-Anwendung zur Verschlüsselung von homophonen Substitutionschiffren: https://www.cryptoprograms.com/other-create/homophonic

onschiffren.[29] Dieser Analysator unterstützt einige Sprachen und kann den eingegebenen Geheimtext automatisch entschlüsseln. Er erlaubt die Nutzung mehrerer CPU-Threads zur schnelleren Analyse des Geheimtextes. Eine manuelle Analyse ist nicht möglich. Das Ausgabefeld zeigt lediglich eine Auswahl der besten Lösungen an. Die Webseite *dCode* bietet eine Anwendung zur Ver- und Entschlüsselung homophoner Substitutionschiffren unter Angabe eines Schlüssels.[30] Dieser Schlüssel kann komfortabel in einer Tabellenansicht editiert werden. Die Ansicht des Klar- und Geheimtextes ist allerdings ebenfalls rudimentär. Zuletzt soll *Boxentriq*[31] von Johan Åhlén genannt werden. Diese Webseite bietet ebenfalls etliche Anwendungen zur Ver- und Entschlüsselung und auch zur automatischen Analyse. Als Beispiel soll dessen Anwendung zur Vigenère-Chiffre genannt werden.[32] Neben einer allgemeinen Beschreibung der Chiffre gibt es in der Anwendung Textfelder zur Ein- und Ausgabe der Texte und einige Optionen zur Text-Behandlung, für die Anzahl der maximalen Iterationen des automatischen Analysators und weiterer Optionen.

Es gibt also einige Anwendungen für die automatische Analyse von klassischen Chiffren, wobei homophone Substitutionschiffren seltener unterstützt werden. Die Möglichkeit einer manuellen Analyse beziehungsweise Unterstützung eines halbautomatischen Algorithmus sind allerdings in den untersuchten Anwendungen nicht gegeben. Auch ist die Darstellung des Geheim- und Klartextes einfach gehalten, so dass zum Beispiel die visuelle Erkennung der einzelnen Homophone einer homophonenen Substitutionschiffre schwer möglich ist. Hingegen gibt es in der, im Zuge dieser Ausarbeitung, entwickelten Anwendung keine Möglichkeit zur automatischen Analyse von homophonen Substitutionschiffren. Der Fokus lag bei der Entwicklung auf der Umsetzung der manuellen Analyse. Daher ist die Darstellung der einzelnen Homophone und die Abbildung auf Klartextsymbole von großer Bedeutung für die Anwendung. Ebenfalls kann innerhalb der entwickelten Anwendung auf verschiedene Weise navigiert werden. Neben Maus- oder Touch-Navigation ist auch die Verwendung der Tastatur für die gängigen Operationen möglich. Ein direkter Vergleich der entwickelten Anwendung mit den untersuchten Anwendungen ist daher schwer möglich. Je nach Anwendungszweck gibt es unterschiedliche Vor- und Nachteile.

[29] CryptoPrograms-Anwendung zur Analyse von monoalphabetischen Substitutionschiffren: https://cryptocrackprogram.github.io/CryptoPrograms/solve/monoalph.html

[30] dCode-Anwendung zur Ver- und Entschlüsselung von homophonen Substitutionschiffren: https://www.dcode.fr/homophonic-cipher

[31] Boxentriq-Homepage: https://www.boxentriq.com

[32] Boxentriq-Anwendung zur Ver- und Entschlüsselung sowie Analyse von Vigenère-Chiffren: https://www.boxentriq.com/code-breaking/vigenere-cipher

7 Fazit

Die entwickelte Anwendung erfüllt alle beschriebenen Anforderungen aus Kapitel 3.1. In Kapitel 4.3 ist die Implementierung der Trennung des Geheimtextes, also die Einteilung in die Homophone, anhand des Parsers ausgiebig erläutert. Der Parser, zusammen mit den beschriebenen Transformationen auf dem erzeugten Ableitungsbaum, ist maßgeblich verantwortlich für die korrekte Verarbeitung der Homophone. Das Kapitel 4.1 zeigt die umgesetzten Anforderungen anhand von Ausschnitten aus der implementierten Benutzeroberfläche. So wird der Ablauf einer Analyse mit Hilfe eines beispielhaften Geheimtextes in seinen verschiedenen Schritten aufgezeigt: Beginnend bei der Eingabe im Geheimtext-Editor, zur Visualisierung der Bestandteile innerhalb der Hauptansicht zusammen mit der Ansicht der Software-Tastatur, bis zur Darstellung des resultierenden Klartextes und Schlüssels.

7.1 Zielerreichung

Dank des Fokus der Anwendung auf der manuellen Analyse von Geheimtexten konnten die Anforderungen erfolgreich, ohne Kompromisse bei der Implementierung, umgesetzt werden. Infolge der regelmäßigen Meetings während der Entwicklung konnten frühe Rückmeldungen die Benutzeroberfläche verbessern und neue Ideen und Wünsche in die Implementierung integriert werden. Die Evaluation (Kapitel 5) hat gezeigt, dass die Anwendung für die manuelle Analyse von homophonen Substitutionschiffren gut geeignet ist. Die Implementierung des Parsers erlaubt, neben der flexiblen und schnelle Einteilung des Geheimtextes in seine Bestandteile, zusätzlich einfache Anpassungen an zukünftige Anforderungen. Wie in Kapitel 4.6 beschrieben, mussten zunächst einige Optimierungen, wie Caching und Windowing, eingeführt werden, damit die Anwendung eine wünschenswerte Geschwindigkeit aufweist. Ohne diese Optimierungen führte bereits die einfache Navigation innerhalb großer Geheimtexte zu Einschränkungen der Benutzbarkeit, auf Grund von vielfältigen, automatischen Neuberechnungen unbeteiligter Komponenten. Zusätzlich wird gezeigt, dass es weiteres Potential gibt, um die Leistungsfähigkeit der Anwendung weiter zu steigern. Sollte die zukünftige Nutzung zeigen, dass regelmäßig sehr große Dokumente verarbeitet werden, können die in dem Kapitel erwähnten zusätzlichen Optimierungen, wie die Anbindung der Benutzeroberfläche an die lazy-Methoden des Parsers, darüber hinaus weiteres Potential der Anwendung zeigen.

7.2 Ausblick

Wie in der Ausarbeitung beschrieben, gibt es einige kleinere Erweiterungen, die die bestehenden Funktionalitäten der Anwendung verbessern können. Erwähnt seien die

weitere Aufbereitung der Darstellung der transkribierten Symbole nach den DECRYPT-Richtlinien zur Transkription, die Implementierung einer *Undo- / Redo*-Funktionalität und einer Suchfunktion sowie die Nutzung der *File System Access API* für Webbrowser und die damit verbundene bessere Projektverwaltung.

Zusätzlich zu diesen kleineren Erweiterungen vorhandener Funktionalität ist die Einbindung einer automatischen Analyse, wie sie schon in CT2 vorhanden ist, eine wichtige, weitreichende Erweiterung. Die automatische Analyse, auch in Verbindung mit den vorhandenen, manuellen Editiermöglichkeiten, erleichtert die Analyse ungemein. Diese automatische Analyse kann die wahrscheinlichen Klartextsymbole zu den Homophonen des Geheimtextes finden. Daneben kann ein automatischer Algorithmus auch andere Aufgaben übernehmen. Zum Beispiel die Markierung von vollständig entschlüsselten Wörtern anhand eines gegebenen Wörterbuchs.

Eine weitere Idee zur Erweiterung ist die Anbindung der Anwendung an eine API der *DECODE*-Datenbank des DECRYPT-Projekts. Eine solche Anbindung könnte die Speicherung des Fortschritts der Geheimtext-Analyse im Zusammenhang mit den untersuchten historischen Dokumenten erlauben.

Literaturverzeichnis

[1] Friedrich L. Bauer. *Entzifferte Geheimnisse*. Springer Verlag Berlin Heidelberg New York, 2000. ISBN: 3-540-67931-6.

[2] *bvaughn/react-virtualized: React components for efficiently rendering large lists and tabular data*. URL: https://github.com/bvaughn/react-virtualized (besucht am 20. 07. 2023).

[3] *Comparison - Chakra UI*. URL: https://chakra-ui.com/getting-started/comparison#the-runtime-trade-off-%EF%B8%8F (besucht am 24. 07. 2023).

[4] The Unicode Consortium. *The Unicode Standard, Version 15.0.0*. Mountain View, CA: The Unicode Consortium, 2022. ISBN: 9781936213320.

[5] *CrypTool-Website rundum erneuert - CrypTool Portal*. 2020. URL: https://www.cryptool.org/de/posts/2020-10-30/cryptool-website-completely-renewed (besucht am 01. 08. 2023).

[6] Mark Davis, Laurențiu Iancu und Christopher Chapman. *Unicode®Standard Annex #29. UNICODE TEXT SEGMENTATION*. 2022. URL: https://www.unicode.org/reports/tr29/tr29-41.html (besucht am 12. 06. 2023).

[7] Check Easttom. *Modern Cryptography*. Springer Cham, 2022. ISBN: 978-3-031-12303-0.

[8] *Emotion - Introduction*. URL: https://emotion.sh/docs/introduction (besucht am 24. 07. 2023).

[9] Bernhard Esslinger, Hrsg. *Das CrypTool-Buch: Kryptographie lernen und anwenden mit CrypTool und SageMath*. 12. Aufl. CrypTool-Projekt, 2018.

[10] David Flanagan. *JavaScript – Das Handbuch für die Praxis*. O'Reilly Media, Inc., 2021. ISBN: 9783960091578.

[11] Heinz-Peter Gumm und Manfred Sommer. *Informatik, Band 3: Formale Sprachen, Compilerbau, Berechenbarkeit und Komplexität*. De Gruyter Oldenbourg, 2019. ISBN: 978-3-11-044238-0.

[12] Nils Hartmann und Oliver Zeigermann. *React - Grundlagen, fortgeschrittene Techniken und Praxistipps*. dpunkt.verlag GmbH, 2020. ISBN: 9783864905520.

[13] David Kahn. *The Codebreakers – The Story of Secret Writing*. Macmillan Publishing Co., Inc., 1967. ISBN: 0-02-560460-0.

[14] Nils Kopal. „Cryptanalysis of Homophonic Substitution Ciphers Using Simulated Annealing with Fixed Temperature“. In: *Proceedings of the 2nd International Conference on Historical Cryptology* 37 (2019), S. 107–116.

[15] Nils Kopal, Olga Kieselmann, Arno Wacker und Bernhard Esslinger. „CrypTool 2.0“. In: *Datenschutz und Datensicherheit-DuD* 38.10 (2014), S. 701–708.

[16] Beáta Megyesi. „Transcription of Historical Ciphers and Keys: Guidelines“. In: (2020). URL: https://www.diva-portal.org/smash/get/diva2:1437998/FULLTEXT01.pdf.

[17] *memo – React*. URL: https://react.dev/reference/react/memo#should-you-add-memo-everywhere (besucht am 17. 07. 2023).

[18] *Optimizing Performance – React*. URL: https://legacy.reactjs.org/docs/optimizing-performance.html (besucht am 17. 07. 2023).

[19] David Salomon. *Coding for Data and Computer Communications*. Springer New York, NY, 2005. ISBN: 978-0-387-21245-6.

[20] Pavel Savara und Luke Latham. *Run .NET from JavaScript | Microsoft Learn*. 2022. URL: https://learn.microsoft.com/en-us/aspnet/core/client-side/dotnet-interop?view=aspnetcore-7.0 (besucht am 13. 07. 2023).

[21] *Styled System*. URL: https://styled-system.com/getting-started (besucht am 24. 07. 2023).

[22] *UCL_Library_Services_Special_Collections_Henry_Brougham_Archive_915uncat*. UCL Library Services - Special Collections. URL: https://de-crypt.org/decrypt-web/RecordsView/6317 (besucht am 16. 08. 2023).

[23] *Was ist eine Webanwendung?* URL: https://aws.amazon.com/de/what-is/web-application/#seo-faq-pairs#what-are-the-benefits-of-web-applications (besucht am 03. 08. 2023).

Abkürzungsverzeichnis

CT	CrypTool
CT2	CrypTool 2
CTO	CrypTool-Online
JS	JavaScript
Wasm	WebAssembly
SPA	Single Page Application
CSS	Cascading Style Sheets
KFG	kontextfreie Grammatik

Tabellenverzeichnis

List of Listings

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 Bachelor- bzw. Masterarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

_____ _____

(Datum) (Unterschrift)

Inhalt der CD / des USB-Sticks

- Masterarbeit als PDF
- Quellcode der Applikation