UNIVERSITÄT SIEGEN

Naturwissenschaftlich Technische Fakultät

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science Informatik

Benutzerfreundliche Web-Applikation zur Verwendung von OpenSSL auf Basis von WebAssembly

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

vorgelegt von	Jan Neumann
Matrikelnummer	1358430
Abgabedatum	10.05.2021

Kurzzusammenfassung

Das Ziel der vorliegenden Bachelorarbeit ist die Realisierung einer Web-Applikation zur Verwendung von kryptografischen Verfahren aus der Open-Source-Software OpenSSL. Das in OpenSSL enthaltene Kommandozeilenprogramm soll in der Web-Applikation auf eine benutzerfreundliche Art integriert werden, um die Kommandos und deren Funktion näher kennenzulernen. Die Grundlage für dieses Vorhaben bietet das Kompilierungsziel WebAssembly. Daraus ergibt sich die Möglichkeit, eine native Anwendung wie OpenSSL rein Client-seitig im Web-Browser auszuführen. In diesem Zusammenhang werden die Herausforderungen näher beschrieben, um nativen Code in eine Web-Browserumgebung zu integrieren. Ebenfalls werden die Gestaltungsoptionen einer möglichen Benutzeroberfläche für die Kommandozeile untersucht und mit Hilfe weiterer Web-Technologien in die Tat umgesetzt.

Abstract

The goal of this bachelor thesis is the realization of a web application using cryptographic methods from the open-source software OpenSSL. The command line program included in OpenSSL is to be integrated in the web application in a user-friendly way in order to get to know the commands and their function more closely. The compilation target WebAssembly provides the basis for this project. This results in the possibility to run a native application like OpenSSL purely at the client side in the web browser. In this context, the challenges to integrate native code into a web browser environment are described in more detail. Likewise, the design options of a possible user interface for the command line are examined and put into practice with the help of further web technologies.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Die stetige Weiterentwicklung von Web-Technologien bringt neue Standards hervor. Einer der neuen Standards ist WebAssembly, wodurch Inhalte im Web-Browser von nahezu nativen Ausführungsgeschwindigkeiten profitieren können. Als ein Kompilierungsziel ermöglicht es unter anderem, Sprachen aus der C-Familie im Web-Browser auszuführen. So können bereits existierende Programme und Bibliotheken im Web plattformunabhängig bereitgestellt werden.

Eine weit verbreitete, in C geschriebene Open-Source-Software ist OpenSSL. Die Software-Bibliothek kommt heute vor allem bei den Verschlüsselungsprotokollen SSL und TLS in Web-Servern zum Einsatz und stellt Schnittstellen für eine Implementierung zur Verfügung. Außerdem bringt OpenSSL ein Kommandozeilenprogramm mit, welches verschiedene Werkzeuge zur Ausführung von kryptografischen Verfahren umfasst. Mit Hilfe von WebAssembly ergibt sich die Möglichkeit, OpenSSL direkt im Web-Browser aufzurufen und Befehle des Kommandozeilenprogramms auszuführen. Diese Arbeit konzentriert sich darauf, eine solche Möglichkeit im Web-Browser zu realisieren und eventuelle Einschränkungen im Browser weitestgehend zu umgehen.

1.2. Ziele

Ziel dieser Bachelorarbeit ist die Implementierung einer Web-Applikation zur Verwendung von kryptografischen Funktionen auf Basis von OpenSSL. Ein wichtiger Bestandteil ist dabei die Untersuchung, wie sich das Kommandozeilenprogramm von OpenSSL zu einem ausführbaren WebAssembly-Format kompilieren lässt.

Die anschließende Realisierung bietet die Grundlage, um OpenSSL in ein Web-Front-End zu integrieren und rein Client-seitig auszuführen. Des Weiteren steht im Fokus dieser Arbeit eine benutzerfreundliche Gestaltung des Kommandozeilenprogramms auf Basis von Web-Technologien und die Integration in die Webseite des weit verbreiteten Open-Source Lernprogramms CrypTool-Online (CTO)[1]. Dadurch werden die Ergebnisse dieser Arbeit auch für die breite Nutzung verfügbar sein und weiter gepflegt werden.

[1] `https://www.cryptool.org/de/cto/`

1.3. Aufbau der Arbeit

Die Bachelorarbeit unterteilt sich in sechs Kapitel, wobei Kapitel 1 die Einleitung darstellt.

Kapitel 2 führt in OpenSSL ein und behandelt die Möglichkeiten, nativen Code im Browser auszuführen. Zudem werden die Technologien zur Umsetzung von Web-Applikationen vorgestellt.

Darauf folgt in Kapitel 3 eine Analyse mit den zu erwartenden Szenarien für die Benutzung einer möglichen Umsetzung von OpenSSL als Anwendung im Web. Dabei werden die Anforderungen beschrieben und die Funktionsweise der Web-Applikation anhand eines Entwurfs erklärt. Zusätzlich werden potentielle Technologien für die Entwicklung der Applikation auf Stärken und Schwächen analysiert.

In Kapitel 4 steht die Implementierung von Komponenten im Fokus, die den Aufbau der Anwendung definieren.

Anschließend wird in Kapitel 5 die Realisierung der Client-seitigen Web-Applikation evaluiert. In diesem Zusammenhang stehen vor allem die Implementierungsziele im Fokus.

Zuletzt folgt in Kapitel 6 eine Zusammenfassung und ein Ausblick über zukünftige Einsatzzwecke und mögliche Erweiterungen.

Alle Abbildungen dieser Arbeit sind mit den Programmen draw.io und LibreOffice erstellt worden oder es wurde die Quelle dazu angegeben.

2. Grundlagen

2.1. Einführung in OpenSSL

Die Basis für das heutige OpenSSL-Projekt wurde 1995 durch Eric A. Young und Tim J. Hudson ins Leben gerufen. Anfangs noch unter dem Namen SSLeay entwickelten Young und Hudson eine Open-Source-Implementierung des Verschlüsselungsprotokolls Secure Sockets Layer (SSL). Im Jahr 1998 wurde SSLeay schließlich als separates Projekt unter dem Namen OpenSSL von der Community fortgeführt, nachdem keine weitere Version mehr von SSLeay erschien. [21]

Heute ist der produktive Einsatz der freien Software-Bibliothek OpenSSL vor allem auf Web-Servern für eine sichere Datenübertragung weit verbreitet. Laut SimilarTech beträgt der Marktanteil von OpenSSL 85,32%. [15] Das nach Federal Information Processing Standard (FIPS) 140-2 zertifizierte Open-Source-Programm lässt sich insgesamt in vier Komponenten unterteilen, wie in Abbildung 1 zu sehen ist. Die Kernkomponente *libcrypto* ist eine Kryptobibliothek bestehend aus einer Vielzahl kryptografischer Primitive. Diese bildet die Grundlage für die weiteren Bestandteile von OpenSSL. Dazu gehört eine Implementierung von SSL, bzw. des Nachfolgerprotokolls Transport Layer Security (TLS) in der Komponente *libssl* und eine sogenannte Engines-Komponente, welche die Funktionalität der Kryptobibliothek durch bspw. Hardware-Geräte erweitern kann. Darauf aufbauend gibt es noch eine Applikationskomponente in Form einer Kommandozeile, welche die Funktionen aus *libcrypto* und *libssl* bereitstellt, wie zum Beispiel Ver- und Entschlüsselung von Daten, Schlüsselerzeugung und SSL/TLS-Werkzeuge.

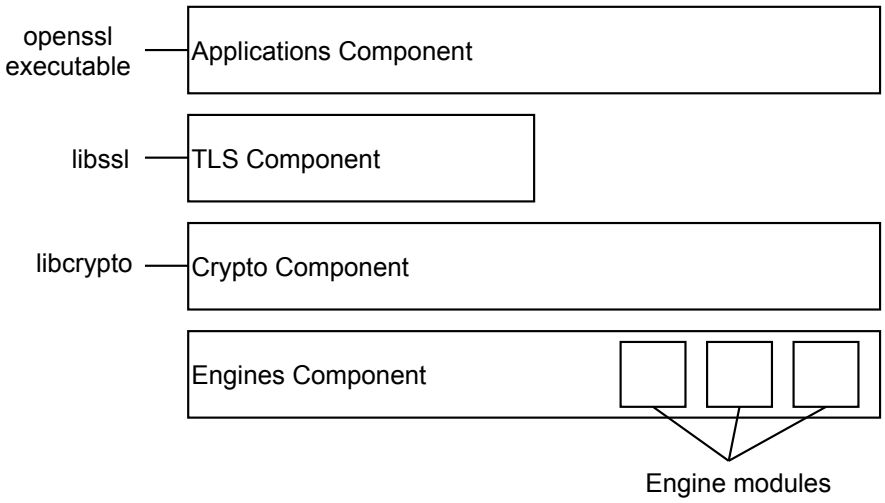

Abbildung 1: OpenSSL Packaging View, in Anlehnung an [18]

Zum Zeitpunkt dieser Arbeit steht OpenSSL 1.1.1k als Long Term Support (LTS) Version bereit. Aufgrund der schon existierenden Benennung des FIPS-Object-Module 2.0, wurde die Hauptversionsnummer 2.x übersprungen und somit ist OpenSSL 3.0 als nächste große Veröffentlichung geplant. [17] Diese steht aktuell als Alpha-Version auf der offiziellen Website zur Verfügung und lässt sich testen. Hinzuzufügen ist zudem noch die Release-Häufigkeit der Open-Source-Software. Alleine 2020 erschienen fünf sogenannte Letter Releases der Version 1.1.1 für Bug- und Sicherheits-Fixes. Das offizielle GitHub-Repository zählt mittlerweile 556 Contributors und insgesamt 28.688 Commits. Damit wird deutlich, wie aktiv die Community an dem Open-Source-Projekt mitwirkt und die Software stetig erweitert und aktualisiert wird. [26]

Zu den grundlegenden Neuerungen der Version 3.0 zählen Änderungen in den APIs und der Lizensierung. Ersteres betrifft die Low-Level APIs, welche fortan als veraltet gelten. Das Entwickler-Team möchte damit das sogenannte Provider-Konzept in den Vordergrund stellen, welches als eine Ansammlung verschiedener algorithmischer Implementierungen dient und nur über High-Level APIs aufgerufen werden kann. Die Low-Level APIs sind hingegen nur auf bestimmte algorithmische Implementierungen ausgerichtet. Einer der Standard-Provider in Version 3.0 ist das FIPS-Modul, welches nun direkt in OpenSSL integriert ist. Zuvor war das FIPS-Modul separat verfügbar und konnte nur in Verbindung mit einer kompatiblen OpenSSL-Version verwendet werden. Die im FIPS-Modul zur Verfügung stehenden kryptografischen Algorithmen sind durch das Cryptographic Module Validation Program (CMVP) nach FIPS-Standards validiert. [16]

Zum Zeitpunkt der Abgabe dieser Arbeit wurde die Release-Version 1.1.1k und die Test-Version alpha15 von 3.0.0 verwendet.

Im Folgenden werden die Funktionen und die Verwendung des Kommandozeilenprogramms näher erläutert. Offiziell bietet die OpenSSL Software Foundation nur den Quelltext einer Version an, welche dann selbst für das jeweilige System mit Hilfe eines Konfigurationskripts individuell angepasst wird und anschließend mit einem C-Compiler kompiliert werden muss. Jedoch werden heute viele Linux-Distributionen mit einer vorkompilierten OpenSSL-Version ausgeliefert. Das hat den Vorteil, dass sich OpenSSL somit direkt in einer Unix-Shell aufrufen lässt. Daher wurden alle nachstehend genannten Kommandos unter Ubuntu 20.04 LTS ausgeführt.

Versions-Abfrage

Ein simples Kommando zur Überprüfung der vorliegenden OpenSSL-Version lautet wie folgt:

```
$ openssl version
OpenSSL 1.1.1k  25 Mar 2021
```

Liste aller verfügbaren Kommandos

Eine Übersicht aller OpenSSL-Kommandos erhält man mit folgendem Befehl:

```
$ openssl help
```

Die Ausgabe unterteilt sich in insgesamt drei Abschnitte mit der Auflistung aller Standard-, Message-Digest- und Cipher-Kommandos. Nähere Informationen zur Verwendung einzelner Kommandos lassen sich mit dem Zusatzargument `-help` ausgeben. In der Ausgabe werden alle Optionen bzw. Argumente gelistet. Diese definieren den Aufbau eines Kommandos wie folgt:

```
$ openssl command [ command_opts ] [ command_args ]
```

RSA-Schlüsselgenerierung

Die Generierung eines RSA-Schlüsselpaars kann auf mehreren Wegen erfolgen. Eine Variante, um einen privaten Schlüssel zu erzeugen, ist das Kommando `genrsa`. Die Verwendung ohne weitere Optionen würde standardmäßig einen privaten Schlüssel der Länge 2048 bit in der Kommandozeile ausgeben. Zusätzlich wird auch ein Status zu jeder Berechnung angezeigt mit Informationen zu Primzahltests. Aufgrund des Zufallsprozesses bei der Schlüsselerzeugung kann die Zeit für eine Berechnung variieren.

```
$ openssl genrsa
```

Die Erzeugung mit `genrsa` kann jedoch auch genauer spezifiziert werden. Möchte man beispielsweise, dass der private Schlüssel als Dateiausgabe erfolgt, lässt sich das durch die zusätzliche Angabe des `-out` Parameters und einem Dateinamen festlegen. Auch die Länge des Modulus lässt sich bestimmen durch eine Angabe der Schlüssellänge als letzte Option des Kommandos. Hier ist zu beachten, dass nur Werte größer gleich 512 erlaubt sind.

```
$ openssl genrsa -out my_rsa.pem 4096
```

Anders als sonst in der Mathematik üblichen Darstellung des RSA-Verfahrens, enthält der erzeugte private Schlüssel auch die Werte des öffentlichen Schlüssels. Mit folgendem Befehl lassen sich alle relevanten Daten des erzeugten Schlüssels anzeigen [8]:

```
$ openssl pkey -in my_rsa.pem -pubout -text
```

Aus dem privaten Schlüssel lässt sich der zugehörige öffentliche Schlüssel mit dem Kommando `rsa` und der Option `-pubout` erstellen. Dafür wird der zuvor generierte private Schlüssel mit dem Input-Parameter `-in` angegeben. Um den öffentlichen Schlüssel ebenfalls als Datei abzuspeichern, benötigt man auch hier die Angabe des `-out` Parameters. Andernfalls wird der öffentliche Schlüssel lediglich in der Kommandozeile ausgegeben.

```
$ openssl rsa -in my_rsa.pem -pubout -out my_rsa.pub
```

Hash-Funktionen

Eine weitere Funktionalität des OpenSSL-Toolkits ist die Verwendung zahlreicher Hash-Funktionen. Eine Liste aller unterstützten kryptografischen Hash-Algorithmen lässt sich mit dem folgenden Kommando ausgeben:

```
$ openssl dgst -list
```

Standardmäßig verwendet das `dgst`-Kommando SHA-256 zur Integritätsprüfung. Möchte man einen anderen Algorithmus verwenden, so gibt man den gewünschten Algorithmus als Option bei der Ausführung an. Alternativ kann der Name der Hash-Funktion selbst als Kommando verwendet werden. Der MD5-Hashwert einer Datei berechnet sich wie folgt:

```
$ openssl dgst -md5 my_file.txt
```

oder

```
$ openssl md5 my_file.txt
```

Symmetrische Verschlüsselung

OpenSSL bietet eine Reihe von verschiedenen Block- und Stromchiffren zur symmetrischen Ver- und Entschlüsselung von Daten. Der entsprechende Schlüssel ist dabei entweder Passwort-basiert oder kann explizit in Form einer Datei angegeben werden. Ähnlich wie auch zuvor, lassen sich alle unterstützten Chiffren mit dem Kommando `openssl enc -list` anzeigen. Block-Chiffren besitzt außerdem verschiedene Betriebsmodi wie Cipher Block Chaining (CBC), Electronic Code Book (ECB), Cipher Feedback (CFB), Output Feedback (OFB) und Counter Mode (CTR).

Am Beispiel der Blockchiffre AES-256 im CBC-Modus wird im Folgenden die Verschlüsselung einer beliebigen Nachricht demonstriert. Dafür wird zunächst ein zufälliger 32-Byte langer Schlüssel generiert und als Datei abgespeichert:

```
$ openssl rand -out my_key.bin 32
```

Als Initialisierungsvektor (IV) wird ebenfalls ein zufälliger Wert erzeugt. In diesem Fall ein 16-Byte langer Hex-Wert. Der IV kann bei der Ver- und Entschlüsselung jedoch nur in der Hex-String-Repräsentation angegeben werden. Daher erfolgt die Ausgabe von `rand` direkt in der Kommandozeile.

```
$ openssl rand -hex 16
8d2facbd1ae4c3bfdf1494b2c0dc9e7f
```

Mit dem zuvor generierten Schlüssel und dem IV lässt sich die Verschlüsselung wie folgt ausführen. Die Ausgabedatei `my_message.txt.enc` enthält den entsprechenden Geheimtext:

```
$ openssl enc -aes-256-cbc -in my_message.txt -out my_message.txt.enc
↪  -kfile my_key.bin -iv 8d2facbd1ae4c3bfdf1494b2c0dc9e7f -pbkdf2
```

Zum Entschlüsseln wird das gleiche Kommando verwendet. Zusätzlich wird die Option `-d` ergänzt und als Input der zuvor erzeugte Geheimtext übergeben.

Das OpenSSL-Toolkit bezeichnet sich nicht ohne Grund als „[...] a swiss army knife for cryptographic tasks, testing and analyzing.“ [26] Man muss dabei jedoch berücksichtigen, dass OpenSSL keine klassischen kryptografischen Verfahren beinhaltet bzw. keine schwachen Parameter empfiehlt, da es eben für den realen und vor allem produktiven Einsatz gedacht ist. Die hier aufgeführten Kommandos sind nur ein kleiner Ausschnitt von dem, was das Toolkit mitbringt, und beschränken sich hauptsächlich auf die Funktionalitäten für die Verwendung der späteren Web-Applikation. Außen vor gelassen wurden bspw. SSL/TLS-Funktionalitäten oder die Erstellung von X.509-Zertifikaten.

2.2. Nativer Code im Web-Browser

JavaScript (JS) galt lange Zeit als die einzige standardisierte Sprache mit Programmierparadigmen, die von allen populären Web-Browsern unterstützt wird. [30] Als eine vom Browser interpretierte Skriptsprache diente JS anfangs nur zur einfachen Interaktion mit HTML-Dokumenten, um bspw. Formulardaten vor dem Absenden zu überprüfen. Am Beispiel von Google Maps zeigt sich jedoch, dass Webseiten mit der Zeit immer dynamischer werden und damit auch ressourcenintensiver. Im Jahr 2008 stellten die drei Web-Browser Chrome, Firefox und Safari ihre eigenen Just-in-time (JIT)-Compiler vor und beschleunigten damit erheblich die Ausführung von JS. [30] Die JIT-Kompilierung

macht es möglich, JS zur Laufzeit zu optimieren und in effizienten Maschinencode zu übersetzen. Allerdings besitzt JS eine dynamische-Typisierung, das heißt, die Typprüfung findet erst zur Laufzeit statt. Das kann dazu führen, dass dynamisch verwendeter Code nicht von einer JIT-Optimierung profitiert und stattdessen interpretiert werden muss, wodurch Leistungseinbußen entstehen. Problematisch scheint hier vor allem die Möglichkeit, die betreffenden Code-Stellen vorab zu identifizieren, um eine entsprechende Optimierung auszunutzen. [10] Das zeigt, dass JS, aber auch dynamische Sprachen im Allgemeinen, nicht dafür konzipiert sind, ähnliche Ausführungsgeschwindigkeiten zu erreichen wie nativ kompilierte Programme.

Das beschränkt den Browser hinsichtlich der Ausführung vieler sehr ressourcenintensiver Berechnungen wie z.B. physikalische Simulationen oder eine Wiedergabe hochauflösender Szenen, die sich allein mit JS nur schwer umsetzen lassen. [28]

2.2.1. Google Native Client

Ein erster Versuch, C/C++-Code in einer Browserumgebung auszuführen, war das Native Client (NaCl)-Projekt des Technologieunternehmens Google LLC im Jahr 2008. Das Ziel war eine sichere Ausführungsumgebung für den Browser bereitzustellen, welche als eine Art 2-Schichten Sandbox die Ausführung von x86-Maschinencode ermöglicht. Dafür muss die NaCl-Runtime als Browsererweiterung installiert werden, welche dann nebenläufig als eigenständiger Prozess läuft. Dieser Prozess arbeitet direkt auf dem Betriebssystem und nutzt dessen Sicherheitsmechanismen. Der NaCl-Code wird innerhalb der Runtime ebenfalls in einer Sandbox ausgeführt, wie in Abbildung 2 zu sehen ist. Das hat den Vorteil, dass heruntergeladene NaCl-Module vor der Ausführung in der Runtime geprüft werden, um eine sichere Ausführung zu gewährleisten. Dafür implementiert NaCl einen eigens entwickelten Disassembler, der mit Hilfe einer sogenannten Software-based Fault Isolation (SFI)-Technik vorab alle Instruktionen des Maschinencodes überprüft, um vorherzusagen, dass jede einzelne Instruktion sicher ausgeführt werden kann. Ist das nicht der Fall, wird das Modul erst gar nicht ausgeführt. [28]

Zur Interaktion mit dem Browser nutzte NaCl anfangs dieselbe Schnittstelle des Netscape Plugin API (NPAPI) wie unter anderem das Browser-Plugin Adobe Flash Player. Aus Sicherheitsgründen in Bezug auf NaCl wurde die Schnittstelle später durch ein eigenes Derivat des NPAPI namens Pepper Plugin API (PPAPI) ersetzt.

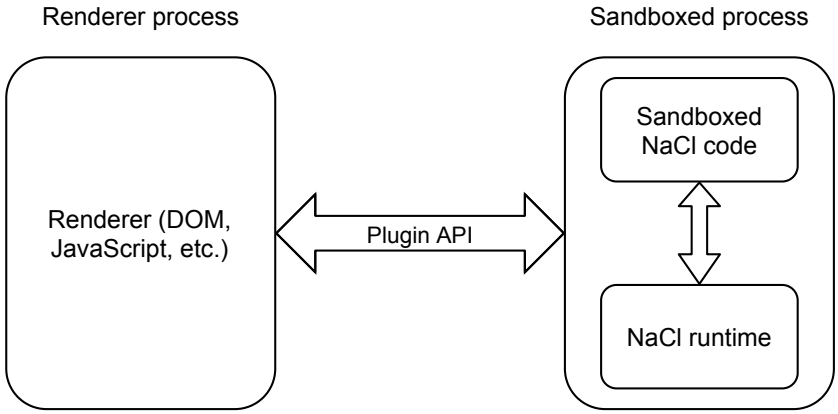

Abbildung 2: Native Client Prozess, in Anlehnung an [30]

Die Performanz (Ausführungsgeschwindigkeit) von NaCl-Modulen ist generell sehr ähnlich zu nativ ausgeführten Programmen. Die Tabelle [2] in Abbildung 3 zeigt einen SPEC2000 Benchmark im Vergleich mit den gemessenen Zeiten in Sekunden. Die Spalte *increase* stellt den entsprechenden Overhead der NaCl-Module gegenüber der Spalte *static* dar. [28]

	static	aligned	NaCl	increase
ammp	200	203	203	1.5%
art	46.3	48.7	47.2	1.9%
bzip2	103	104	104	1.9%
crafty	113	124	127	12%
eon	79.2	76.9	82.6	4.3%
equake	62.3	62.9	62.5	0.3%
gap	63.9	64.0	65.4	2.4%
gcc	52.3	54.7	57.0	9.0%
gzip	149	149	148	-0.7%
mcf	65.7	65.7	66.2	0.8%
mesa	87.4	89.8	92.5	5.8%
parser	126	128	128	1.6%
perlbmk	94.0	99.3	106	13%
twolf	154	163	165	7.1%
vortex	112	116	124	11%
vpr	90.7	88.4	89.6	-1.2%

Abbildung 3: Native Client Benchmark [28, S.86]

Problematisch war vor allem die Beschränkung auf x86-Maschinencode und die damit zusammenhängende Portabilität der NaCl-Module. Besonders im Web ist es umso wichtiger, dass Inhalte unabhängig von Web-Browser, Betriebssystem oder der verwendeten Hardware aufrufbar sind. Um diesem Problem entgegenzuwirken, wurde das NaCl-Projekt dahingehend erweitert, dass die NaCl-Module zuerst als LLVM Intermediate

[2] In diesem Vergleich wurden alle Binärdateien statisch gelinkt. Die Spalte *aligned* zeigt zusätzlich die Ausführung mit einer geänderten Speicherausrichtung des Binärdateien an. [28]

Representation (LLVMIR) vorliegen und folglich Client-seitig für die jeweilige Zielplattform kompiliert werden. [30]

Trotz all der Bemühungen seitens Google, die NaCl-Ausführungsumgebung als einen Standard für alle Browser zu etablieren, konnte sich das Projekt nicht wirklich durchsetzen. Ein Hauptgrund dafür war zu dieser Zeit die allgemeine Abneigung der Browser-Hersteller gegenüber Plugins. Zum einen aufgrund von Sicherheitsbedenken, wie es bei Adobe Flash Player in der Vergangenheit immer wieder der Fall war. [1] Zum anderen beschränken Plugins die Zugänglichkeit von Web-Inhalten, da vorausgesetzt wird, dass der Benutzer das jeweilige Plugin installiert hat. Letztlich verkündete Google im Jahr 2017 zugunsten von WebAssembly, dass das Projekt eingestellt wird und fortan als veraltet gilt.

2.2.2. Emscripten und asm.js

Parallel zum NaCl-Projekt von Google entstand 2010 das Projekt Emscripten durch Alon Zakai. Emscripten fungiert im Unterschied zu NaCl als eine Compiler-Toolchain, die Low Level Virtual Machine (LLVM)-Assembly entgegennimmt und dieses Format zu JavaScript umwandelt. Das Compiler-Projekt LLVM dient dazu, Sprachen wie C, C++ oder Objectiv-C zuerst über ein Compiler-Front-End in das sogenannte LLVMIR-Format zu kompilieren, woraus anschließend durch ein Compiler-Back-End ein entsprechender Maschinencode für eine bestimmte Rechnerarchitektur generiert werden kann. Emscripten nimmt die Rolle eines solchen Back-Ends ein, jedoch mit dem Unterschied, dass eine Low-Level-Sprache in eine High-Level-Sprache umgewandelt wird. Das bringt eine Reihe von Herausforderungen mit sich. Eine Low-Level-Repräsentation besitzt nämlich keine Informationen über die eigentliche Struktur der High-Level-Sprache, wie Schleifen oder If-Anweisungen. Diese Struktur muss aus den einzelnen Code-Fragmenten der LLVMIR rekonstruiert werden. [29]

Da das Kompilierungsziel JavaScript ist, braucht es Browser-seitig auch keine Erweiterung und kann im Gegensatz zu NaCl nativ im Browser ausgeführt werden. Dabei sollte jedoch nicht außer Acht gelassen werden, welche Einschränkungen die JS-Engine im Browser mit sich bringt. Ein Beispiel dafür ist Multithreading. Zwar bieten die Browser sogenannte Web Workers an, die als Threads nebenläufig zum Main-Thread des Browsers laufen, jedoch gibt es keinen gemeinsamen Zustand, da die Web Workers über Nachrichten kommunizieren. [29] Auch Emscripten bietet mittlerweile den Support für Multithreading an, aber nur auf Basis des SharedArrayBuffer in JavaScript, welcher aktuell noch von den Web-Browsern standardisiert und implementiert werden muss. [20]

Die Performanz des Emscripten JS-Codes ist zum Teil deutlich langsamer als eine native Ausführung von C++-Programmen aufgrund der JIT-Kompilierung. Die Tabelle in Abbildung 4 zeigt einen Benchmark, der zum einen die JS-Engines SpiderMonkey (Firefox)

und V8 (Chrome) vergleicht, zum anderen den Referenzwert der nativen Ausführung angibt. Die Spalte *ratio* gibt in dem Fall das Verhältnis an, wie viel langsamer der (fett gedruckte) JS-Code im Vergleich ist.

benchmark	SM	V8	gcc	ratio
fannkuch (10)	1.158	**0.931**	0.231	4.04
fasta (2100000)	**1.115**	1.128	0.452	2.47
primes	**1.443**	3.194	0.438	3.29
raytrace (7,256)	**1.930**	2.944	0.228	8.46
dlmalloc (400,400)	5.050	**1.880**	0.315	5.97

Abbildung 4: Emscripten Benchmark [29, S.7]

Generell lässt sich sagen, dass die Ausführungsgeschwindigkeiten immer noch gut sind und für die meisten Zwecke ausreichen. Einer der Hauptgründe für Emscripten ist die Portabilität von Programmcode.

Die JIT-Kompilierung in JavaScript kann unzuverlässig sein durch die Verwendung dynamischer Typen, was sich letztlich in der Ausführungsgeschwindigkeit bemerkbar macht. Um dieses Problem zu umgehen, wurde im Jahr 2013 asm.js von Luke Wagner und David Herman ins Leben gerufen. Asm.js lässt sich beschreiben als eine beschränkte Teilmenge von JavaScript mit zusätzlichen Konventionen, die zu einem statischen Typsystem führen. Dafür wird der JS-Code so umgeschrieben, dass die jeweiligen Code-Stellen einen statischen Typ implizieren. Normalerweise wird asm.js nicht direkt vom Benutzer selbst geschrieben, sondern dient vielmehr als ein Kompilierungsziel und wird für die Performanz optimiert. Im Zusammenspiel mit Emscripten als Compiler-Toolchain lassen sich somit C/C++-Programme direkt zu asm.js kompilieren.

Letztlich ist asm.js auch nur ein Workaround für den Browser. Denn am Ende ist es immer noch JavaScript in einer Textform, die vor der Ausführung vom Browser geparst werden muss. Das kostet, im Vergleich zu einem effizienten Binärformat, deutlich mehr Zeit. [30] Auch asm.js gilt zugunsten von WebAssembly mittlerweile als veraltet.

2.2.3. WebAssembly

Aufgrund der nicht konformen Web-Standards von Google Native Client und asm.js, entschlossen sich die Browser-Hersteller und das World Wide Web Consortium (W3C) im Jahr 2015, gemeinsam an einer zusätzlichen standardisierten Sprache für den Web-Browser zu arbeiten. Nach zweijähriger Entwicklung erschien WebAssembly 1.0 als Minimum Viable Product (MVP). Es kombiniert die Vorteile der vorherigen Ansätze durch NaCl und asm.js. Noch im selben Jahr erhielt WebAssembly als offener Standard die Unterstützung in allen großen Browsern.

WebAssembly (Wasm) ist ein Low-Level Bytecode, der sich durch seine Kompaktheit auszeichnet und damit wenig bis keinen Overhead bei einer Ausführung besitzt. Beim Designprozess wurde darauf geachtet, dass der Bytecode von Wasm plattformunabhängig und damit an keine spezifische Hardware bzw. Plattform gebunden ist. Eine Eigenschaft, die nicht außer Acht gelassen werden darf, ist die Sicherheit. Vor allem im Web ist der Aufruf unsicherer Quellen nicht ungewöhnlich. Aus diesem Grund ist die Wasm-Runtime eine Virtuelle Maschine (VM), die in einer Host-Applikation ausgeführt wird. [11]

Auch Wasm wird in der Regel nicht selbst geschrieben, sondern dient als ein Kompilierungsziel. Dementsprechend wurde die Compiler-Toolchain Emscripten erweitert, um C/C++-Programme in das Wasm-Binärformat zu kompilieren. Dabei gibt es zusätzlich die Option einen JS-Glue-Code zu generieren, um die Funktionalitäten des erzeugten Wasm-Moduls in einer Browserumgebung zu unterstützen. Die dort enthaltenen Runtime-Methoden dienen als Schnittstelle zu den Implementierungen von C/C++-Bibliotheken wie der Standardbibliothek, SDL, OpenGL oder OpenAL. Ungeachtet dessen besteht auch die Möglichkeit Wasm-Module selbst zu schreiben in einer rein textuellen Form. Dieses Textformat kann daraufhin zu einem Binärformat kompiliert werden.

Ein großer Vorteil von Wasm ist die native Unterstützung des Browsers. Anders als bei NaCl benötigt Wasm keine Browsererweiterung für die Ausführung. Das Wasm-Modul wird mittels der WebAssembly JavaScript-API in einer Webanwendung instanziiert und ausgeführt. Das Code-Listing 1 zeigt beispielhaft den Aufruf eines Wasm-Moduls. Die Methode `instantiateStreaming()` ist zuständig für das Abrufen, Kompilieren und Instanziieren des Wasm-Moduls. Wichtig ist, dass WebAssembly keinesfalls JavaScript ersetzen soll, sondern dafür dient, anspruchsvolle Berechnungen einer Web-Anwendung zu übernehmen.

```
1 WebAssembly.instantiateStreaming(fetch('simple.wasm'))
2 .then(obj => {
3   // 'obj' contains all exported functions
4   obj.instance.exports.exported_func();
5 });
```

Listing 1: Wasm-Modul Instanziierung in JavaScript

Der Vergleich zu JavaScript bezüglich der Performanz macht dies auch deutlich. Die Abbildung 5 zeigt den Speedup-Gewinn als Faktor, der durch die Ausführung von Wasm im Vergleich zu JS erzielt wird. Dafür werden unterschiedliche Hardware und Plattformen in Betracht gezogen. Ein Beispiel das besonders heraussticht, ist der Microsoft Edge Browser unter Windows. Dort ist die Ausführungsgeschwindigkeit von Wasm rund 2,5-mal höher im Vergleich zu JS.

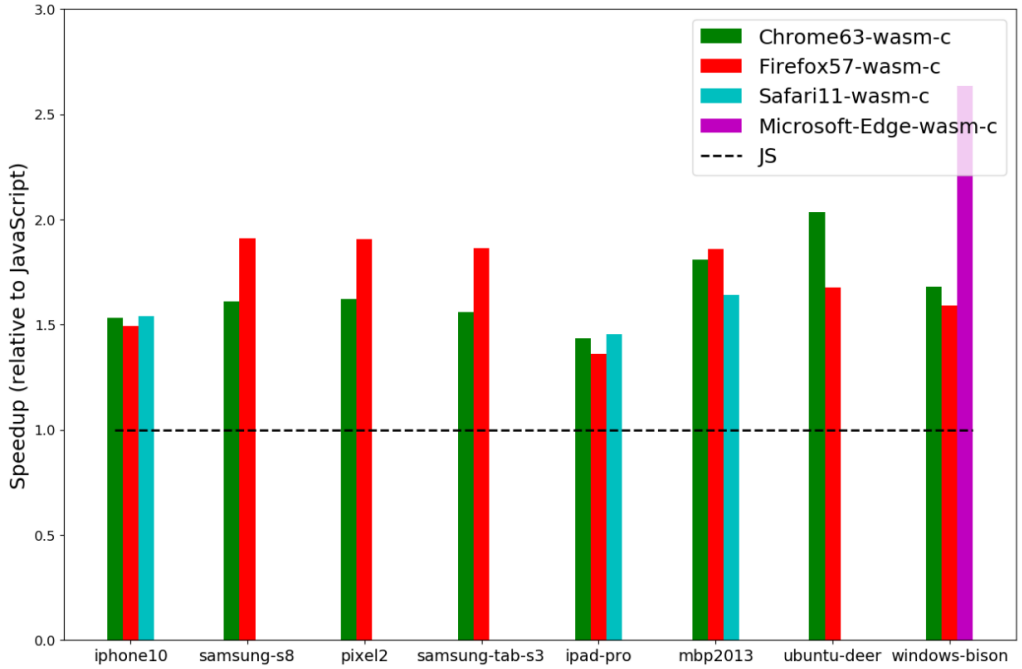

Abbildung 5: Wasm Performanz im Vergleich zu JS [12, S.15]

Mittlerweile beschränkt sich die Ausführung von WebAssembly jedoch nicht mehr nur auf eine Browserumgebung. Seit der Veröffentlichung wurden weitere Entwicklungen ins Leben gerufen wie das WebAssembly System Interface (WASI), die eine standardisierte API für systemorientierte Funktionen bereitstellt, ähnlich wie das Portable Operating System Interface (POSIX). Hierbei möchte man die Vorteile von WebAssembly wie Portabilität und Sicherheit nutzen.

2.3. Entwicklung von Web-Applikationen

Das Web ist für Softwarehersteller eine attraktive Zielplattform, denn die Benutzer brauchen für die Verwendung ihrer Software-Applikation lediglich eine Internetverbindung und Zugriff auf einen Web-Browser. In den meisten Fällen ist die Webseite dann auch gleichzeitig für die mobile Verwendung optimiert. Somit erreicht man mit einer Code-Basis eine Vielzahl von Geräten und Plattformen. Heute finden sich im Web daher nicht nur die klassischen Webseiten wieder, wie Firmenpräsenzen, Blogs oder Newsportale, sondern auch vollständige und komplexe Anwendungen, welche aus dem Desktop-Bereich ins Web portiert wurden. Beispiele dafür sind die Office-Anwendungen von Microsoft[3] oder Google Docs[4]. Aufgrund der Komplexität solcher Anwendungen hat sich dementsprechend auch die Art und Weise verändert, wie Web-Applikationen heutzutage

[3] `https://www.office.com/`
[4] `https://www.google.de/intl/de/docs/about/`

entwickelt werden. Im Folgenden wird darauf eingegangen, aus welchen Komponenten heute eine Applikation im Web besteht und es werden Möglichkeiten zu einer besseren Skalierung solcher Applikationen erläutert.

2.3.1. Web-Technologien

Eine Web-Applikation ist eine Client-Server-Anwendung, die sich über den Einsatz verschiedener Web-Technologien definiert. Dabei unterteilt sich die Web-Applikation in mehrere Schichten. Zum einen in eine Client-Schicht, wo die Darstellung der Anwendung in der Regel im Browser erfolgt. Prinzipiell ist auch eine textuelle Darstellung einer Anwendung bspw. über die Kommandozeile möglich. Darüber hinaus gibt es eine Web-Schicht, in der sich ein Web-Server um die die Bereitstellung und Verarbeitung von Daten der Web-Applikation kümmert. Die Kommunikation zwischen Client und Server erfolgt dabei auf Basis von HTTP bzw. HTTPS. Zudem kommuniziert der Web-Server gleichzeitig mit einer Daten-Schicht, die für die Persistenz der Daten zuständig ist. [22]

Auch die Verlagerung der Anwendungslogik auf den Client ist heute nicht mehr unüblich. Die sogenannten Single Page Applications (SPAs) werden dynamisch durch JavaScript im Browser aufgebaut und kommunizieren im Hintergrund mit dem Web-Server. [22] Die nachstehende Abbildung 6 zeigt den Aufbau der oben beschriebenen 3-Schichten-Architektur.

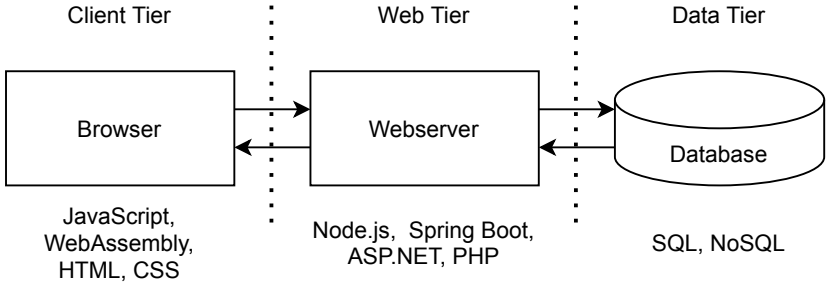

Abbildung 6: 3-Tier-Architektur einer Webanwendung, in Anlehnung an [22, S.2]

Gegenstand dieser Arbeit ist nur die Client-Schicht. Infolgedessen werden die Clientseitigen Web-Technologien zur Darstellung einer Anwendung näher erläutert.

HTML

Hypertext Markup Language (HTML) ist eine Auszeichnungssprache und der grundlegendste Baustein des Web. Als ein Dokument beschreibt HTML das Layout einer Webseite und strukturiert gleichzeitig dessen Inhalt. Der Begriff *Hypertext* bezieht sich dabei auf die Links innerhalb eines HTML-Dokuments, um mehrere Dokumente miteinander zu verknüpfen. Die Auszeichnungssprache verwendet spezielle HTML-Elemente für die Beschreibung einer Webseite, welche in einer Baumstruktur angeordnet werden. Der grundlegende Aufbau eines solchen Dokuments ist in Listing 2 zu sehen.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Titel</title>
    <link rel="stylesheet" type="text/css" href="style.css" />
    <script src="script.js"></script>
  </head>
  <body>
    <p>Inhalt</p>
  </body>
</html>
```

Listing 2: Beispiel eines HTML-Dokuments

Am Anfang jedes Dokuments steht die Dokumentendeklaration, um den Typ des Dokuments festzulegen. Innerhalb des `<html>`-Tags unterteilt sich das Dokument in einen HTML-Head und HTML-Body. Ersteres beinhaltet nur Dokumenten-spezifische Informationen wie bspw. verwendete CSS-Stylesheets oder JS-Dateien. Der Anzeigebereich für den Browser wird im HTML-Body deklariert. Dort finden sich unterschiedliche HTML-Elemente für verschiedene Überschriften z.B. `<h1>`, einen Absatz `<p>`, Buttons `<button>` oder Input-Felder `<input>`. Die einzelnen HTML-Elemente bieten außerdem weitere Attribute innerhalb des Tags, womit bspw. bei Hyperlinks mit `href` das Ziel des Links definiert werden kann.

CSS

Um das Erscheinungsbild der einzelnen HTML-Elemente zu bestimmen, wird die Beschreibungssprache Cascading Style Sheets (CSS) verwendet. Das Ziel ist die Trennung von Darstellungsvorgaben und der semantischen Struktur des HTML-Dokuments. [3, S. 45] Die entsprechenden Styles können dabei entweder in einer separaten Datei, im HTML-Head oder im jeweiligen HTML-Element definiert werden. Für die ersten zwei genannten Möglichkeiten werden die HTML-Elemente mittels CSS-Regeln angesprochen.

Dies geschieht über Selektoren, die darüber entscheiden, für welche Elemente das Styling im HTML-Dokument angewendet wird.

```
p {
    font-family: arial, sans-serif;
    font-size: 12px
}
```

Listing 3: Beispiel einer CSS-Regel

Das Beispiel in Listing 3 zeigt eine CSS-Regel, die für alle `<p>`-Elemente im HTML-Dokument aufgrund des Typselektors angewendet wird. Individuelle HTML-Elemente können mit der eigenen `id` über ID-Selektoren angesprochen werden. Erst mit Hilfe der CSS-Regeln können Webseiten dynamisch gestaltet werden. Diese machen CSS insgesamt zu einem sehr mächtigen Werkzeug.

JavaScript

Wie bereits im Kapitel 2.2 erwähnt, handelt es sich bei JavaScript um eine Skriptsprache zur Gestaltung von interaktiven Anwendungen im Browser. Demzufolge verfügt JS über die Möglichkeit, das Document Object Model (DOM) zu manipulieren. Dieses Objekt stellt die Schnittstelle zwischen HTML und JS dar. Die gesamte Baumstruktur eines HTML-Dokuments wird im DOM abgebildet und in verschiedene Knotenelemente aufgeteilt. Wird ein Skript bspw. wie in dem Listing 2 eingebunden, erhält es über die Web API Zugriff auf das DOM und damit auf die verschiedenen Knotenelemente. Mit dem `document`-Objekt in JS lässt sich daraufhin das DOM programmatisch verändern.

Um sich eine Referenz auf ein jeweiliges HTML-Element zu holen, steht die Methode `querySelector()` zur Verfügung, die einen CSS-Selektor entgegennimmt.

```
const btn = document.querySelector('button');
```

Listing 4: Referenz eines HTML-Elements in JavaScript

Mit dieser Referenz ist man nun in der Lage die Darstellung anzupassen, einen Eventlistener hinzuzufügen oder die Referenz aus der DOM-Baumstruktur zu entfernen.

```
function showMessage() {
    alert('Button clicked!');
}
btn.addEventListener('click', showMessage);
```

Listing 5: Hinzufügen eines Click-Eventlisteners

Die Liste[5] aller Eigenschaften und Methoden des `document`-Objekts ist umfangreich und bietet damit viel Spielraum für die dynamische Anpassung einer Webseite.

2.3.2. Front-End-Frameworks

Für komplexere Web-Applikationen im Front-End kann die Verwendung von reinem HTML, CSS und JS ab einem bestimmten Punkt unübersichtlich werden. Das kann damit enden, dass der Code je nach Projektgröße nicht mehr wartbar ist. Software-Frameworks adressieren genau dieses Problem, um ein Projekt nach einer gegebenen Design-Philosophie umzusetzen. Auch im Bereich der Web-Front-Ends, gibt es eine Auswahl an Frameworks, die dazu verhelfen, den Entwicklungsprozess einfacher zu gestalten. Zu den bekanntesten Front-End-Frameworks zählen Angular, React und Vue.js. Zwar haben alle drei unterschiedliche Ansätze, verfolgen jedoch dieselben Ziele bezüglich der Entwicklung einer Benutzeroberfläche im Web.

Das Design der oben genannten Frameworks zielt vor allem darauf ab, Komponenten im gesamten Front-End wiederzuverwenden, um damit den Entwicklungsaufwand zu senken. Die Entwicklung wird begünstigt durch gegebenen Boilerplate-Code, der dabei hilft, immer wiederkehrende Funktionalitäten an beliebigen Code-Stellen aufzurufen. React legt den Entwicklern nahe, das UI einer Anwendung in kleine Komponenten aufzuteilen, um damit die Wiederverwendbarkeit von Code zu fördern. [24] Daraus entsteht eine sogenannte Komponenten-Hierarchie, die dazu beiträgt, dass jede einzelne Komponente einen eigenen Aufgabenbereich hat. Man folgt damit dem Design-Prinzip der *Separation of Concerns*. [14]

Eine weitere wichtige Eigenschaft der Frameworks, ist die Wartbarkeit der Codebasis zu garantieren. Dadurch dass die Anwendung aus unabhängigen Komponenten besteht, haben Änderungen innerhalb einer Komponente keine Auswirkung auf deren Verwendung in der Web-Applikation. Insgesamt wird die Codebasis im Zuge dessen überschaubarer.

[5] `https://developer.mozilla.org/de/docs/Web/API/Document`

3. Entwurf und Analyse der Web-Applikation

In diesem Kapitel werden zunächst mögliche Szenarien für die Umsetzung des Kommandozeilenprogramms von OpenSSL als Web-Applikation erläutert. Dabei wird analysiert, inwiefern das Kommandozeilenprogramm für den Browser geeignet ist. Daraufhin werden die Anforderungen an die Client-seitige Web-Applikation spezifiziert, die sich in funktionale und nicht-funktionale Anforderungen unterteilen. Hier werden die Funktionalitäten näher auf ihre Verwendung im Web-Browser beschrieben. Im Anschluss folgt ein Entwurf der Web-Applikation, welcher näher auf den Aufbau und die Funktionsweise einzelner UI-Komponenten eingeht. Abschließend geht es um eine technische Analyse bezüglich der Auswahl des Softwarestacks zur Umsetzung der Anwendung.

3.1. Szenarien

Wie im Einführungskapitel zu OpenSSL in 2.1 beschrieben, ist der Server-seitige Einsatz der Krypto- bzw. TLS-Bibliothek von OpenSSL für eine gesicherte Datenübertragung weit verbreitet. Das ausführbare Programm *openssl* hingegen ist nur für die Verwendung auf der Kommandozeile (Command Line Interface (CLI) konzipiert, was scheinbar im Widerspruch steht zu einer Verwendung in einer grafischen Benutzeroberfläche. Die Web-Anwendung soll jedoch beides, GUI und Kommandozeile, unterstützen, damit man die Wahl hat und damit ein didaktischer Effekt entsteht, die Kommandozeilenbefehle auch indirekt zu erlernen.

Die nachfolgenden Szenarien erläutern zum einen die Umsetzung des Kommandozeilenprogramms *openssl* in ein Graphical User Interface (GUI), zum anderen aber auch die Umsetzung als eine Kommandozeile im Web mit den Technologien HTML, CSS und JS. Als Grundlage dafür dient die Annahme, dass OpenSSL zu WebAssembly kompiliert werden kann und damit in einer Browserumgebung aufrufbar ist. Zudem wird vorausgesetzt, dass die Ausführung ausschließlich Client-seitig stattfindet.

3.1.1. Grafische Benutzeroberfläche im Web

Bereits in der Vergangenheit gab es durchaus Projekte mit dem Ziel, die Funktionalitäten des OpenSSL-Toolkits als GUI-Anwendung für den Desktop-Bereich umzusetzen. Eines dieser Projekte ist *Pyrite*[6]. Das mittlerweile nicht mehr fortgeführte Python-Programm verwendet OpenSSL und GNU Privacy Guard (GnuPG) als eine Art Back-End, um über die eigene GUI entsprechende Verschlüsselungsoperationen durchzuführen. Ein Nachteil

[6] `https://github.com/ryran/pyrite`

der *Pyrite*-Anwendung ist allerdings die Voraussetzung, dass OpenSSL bereits vorinstalliert sein muss. Zudem ist die Anwendung auch nur mit Linux-Distributionen kompatibel und nutzt das veraltete Python 2.

Das Konzept von *Pyrite* lässt sich in den Web-Browser übertragen, um somit die Vorteile einer Web-Applikation zu nutzen. Demzufolge wäre die Anwendung ohne eine zusätzliche Installation plattformübergreifend im Web erreichbar (vorausgesetzt man hat eine Internetverbindung).

Über GUI-Elemente werden die einzelnen Optionen bzw. Argumente eines OpenSSL-Kommandos ausgewählt. Bestätigt man in der Anwendung die Eingaben zur Ausführung, werden diese intern verarbeitet und zu einem gültigen Kommando zusammengebaut. Schließlich wird das Kommandozeilenprogramm *openssl* mit dem jeweiligen Kommando aufgerufen und erzeugt eine Ausgabe, welche wiederum in das User Interface übertragen werden muss.

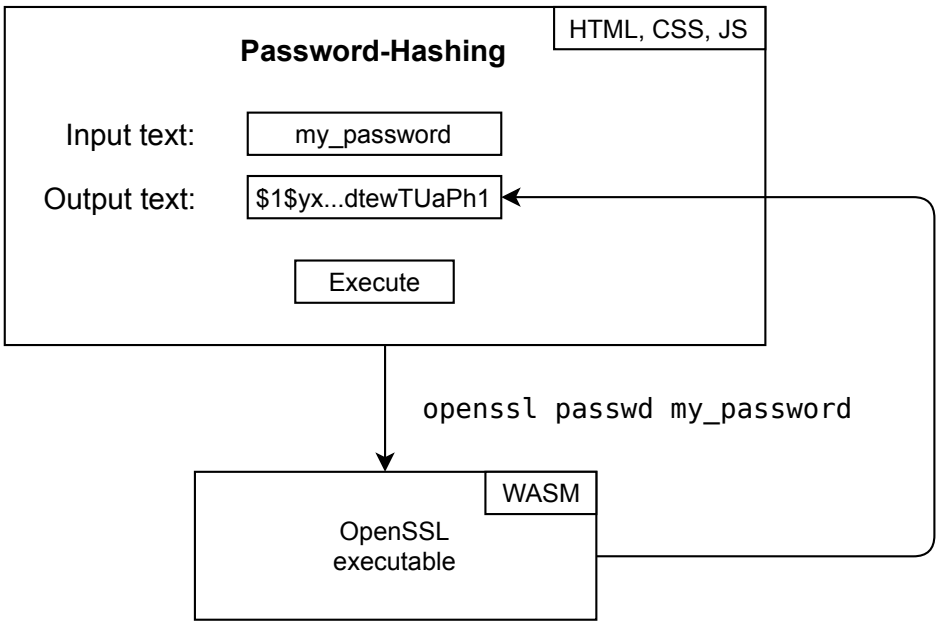

Abbildung 7: Beispielhafte Darstellung einer OpenSSL GUI im Web

In Anbetracht der Umsetzung einer GUI und der entsprechenden Logik, stellt dies für die Web-Technologien HTML, CSS und JS kein Problem dar. Die Abbildung 7 zeigt beispielhaft das Szenario des `passwd`-Kommandos. Nach der Eingabe des Passworts und dessen Bestätigung, erfolgt die Anzeige des Hash-Werts. Die Idee ist, dass jenes Kommando ausgeführt wird, welches der Benutzer sonst über die Kommandozeile aufrufen würde. Prinzipiell lässt sich somit jeder Befehl durch GUI-Elemente interpretieren. Dabei ist entscheidend, dass der Befehl in ein gültiges OpenSSL-Format umgewandelt wird.

Da ein Großteil der Kommandos in OpenSSL für die Input- bzw. Output-Parameter einen Dateipfad erwartet, soll der Benutzer in der Lage sein, über die GUI eine Dateiauswahl zu treffen, die analog dazu im Kommando verwendet wird. Dafür benötigt die Web-Applikation einen direkten Zugriff auf ein Dateisystem. Das erscheint zunächst problematisch angesichts der Browserumgebung, die den uneingeschränkten Zugriff auf das native Client-Dateisystem nicht vorsieht. [9] Abhilfe schaffen in dem Fall virtuelle Dateisysteme innerhalb des Browsers, die zur Laufzeit einer Anwendung ein Dateisystem emulieren. Exemplarisch hierfür ist die File System API von Emscripten. [9]

Wie man die Darstellung beliebig vieler Kommandos als grafische Benutzeroberfläche umsetzt, ist eine Frage des Front-End-Designs. Die Möglichkeiten sind letztlich nur begrenzt durch den Umfang des OpenSSL-Kommandozeilenprogramms.

3.1.2. Kommandozeile im Web

Ein weiteres Szenario ist die Umsetzung als Kommandozeile wie in Abbildung 8. Dieser Ansatz hat zum Ziel, das Verhalten der eigentlichen OpenSSL-Befehlszeile nachzustellen, um ein gleiches Nutzungserlebnis im Web zu bieten. Hier ist allerdings zu unterscheiden, dass es sich dabei nur um eine reine Darstellung mit einer Input-Zeile für das Kommando handeln soll. Weitere Shell-Funktionalitäten werden außen vor gelassen.

Der Benutzer soll in der Lage sein, das Kommando über die Befehlszeile abzuschicken. Anders als bei dem Ansatz der grafischen Benutzeroberfläche, kann die textuelle Eingabe des Kommandos direkt an das OpenSSL-Programm weitergeleitet werden. Dort erfolgt die Ausführung und erzeugt eine Ausgabe, die unter der dargestellten Kommandozeile erscheint.

In diesem Szenario gibt es außerdem dieselbe Problematik bezüglich des Zugriffes auf das lokale Client-Dateisystem. Die Dateien müssen vorab in den Browser geladen werden. Unter Angabe des Dateinamens im Kommando wird die Datei mit Hilfe eines virtuellen Dateisystems verarbeitet.

Mit diesem Ansatz müssen, im Unterschied zu einer grafischen Benutzeroberfläche, keine zusätzlichen Benutzerschnittstellen um das eigentliche Kommando herum implementiert werden. Das Szenario zielt darauf ab, dass prinzipiell alle OpenSSL-Befehle unterstützt werden.

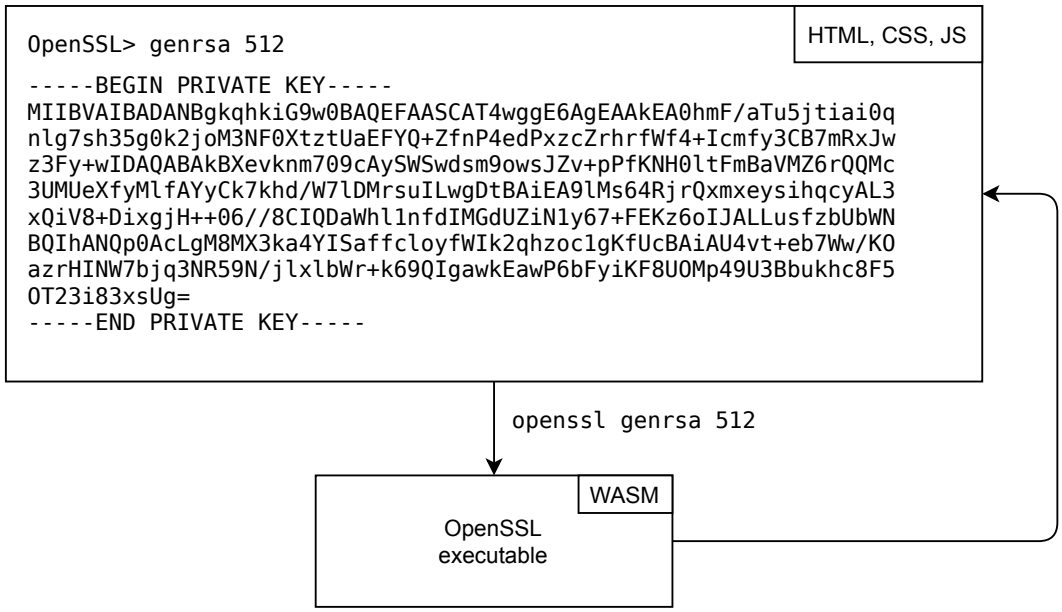

Abbildung 8: Beispielhafte Darstellung einer OpenSSL-Kommandozeile im Web

3.2. Anforderungen

Die Anforderungen der Web-Applikation orientieren sich an den vorgestellten Szenarien einer möglichen Umsetzung. Darüber hinaus stellt das Programm *Pyrite* die Mindestanforderungen an Funktionalitäten, die in der Web-Applikation umgesetzt werden sollen. Die funktionalen und nicht-funktionalen Anforderungen werden im Folgenden näher erläutert.

3.2.1. Funktionale Anforderungen

Zu den Funktionalitäten von *Pyrite* auf Basis von OpenSSL zählt die text- und dateibasierte symmetrische Ver- bzw. Entschlüsselung. Die Web-Applikation soll diese Funktionalitäten um weitere kryptografischen Aufgaben erweitern für eine benutzerfreundlichere Verwendung des OpenSSL-Toolkits. Infolgedessen werden die funktionalen Anforderungen sowohl für eine grafische Benutzeroberfläche als auch für eine Kommandozeile spezifiziert.

Ver- und Entschlüsselung

Der Benutzer soll in der Lage sein, symmetrische Ver- und Entschlüsselungen anhand von Eingabetexten und -dateien durchzuführen. Als Verschlüsselungsverfahren stehen die Algorithmen zur Verfügung, die durch OpenSSL unterstützt werden. Dafür soll zunächst ausgewählt werden, ob der Benutzer ver- bzw. entschlüsseln möchte. Die Web-

Applikation bietet daraufhin eine Reihe von Optionen, die sonst als Befehlsargumente angegeben werden müssten. Dazu zählen:

- Eingabe des Datums
- Auswahl an unterstützten Chiffren
- Ausgabe als Datei und deren Ausgabeformats
- Angabe einer Passphrase (als Text oder Datei)
- Angabe eines Initialisierungsvektors
- eine Key Derivation Function (KDF) anzuwenden

Anschließend soll das Kommando durch eine Bestätigung mit den zuvor ausgewählten Optionen ausgeführt werden. Zur Verdeutlichung, wie das ausgeführte Kommando aufgebaut ist, erfolgt zudem die Anzeige in einem Befehlszeilenformat.

RSA-Schlüsselgenerierung

Die Web-Applikation soll dem Benutzer die Möglichkeit bieten, ein RSA-Schlüsselpaar zu erzeugen. Die Ausgabe der privaten und öffentlichen Schlüssel erfolgt als Datei. Zur Erzeugung stehen folgende Optionen bereit:

- Benennung der Ausgabedatei für den privaten Schlüssel
- Auswahl vordefinierter Schlüssellängen

Nach Bestätigung der Ausführung soll das entsprechende OpenSSL-Kommando im Befehlszeilenformat erscheinen. Daraufhin steht eine weitere Option zur Verfügung, um den öffentlichen Schlüssel zu generieren. Generell muss die Möglichkeit gegeben sein, beide Schlüssel anzeigen zu lassen.

Hashing

Der Benutzer soll die Möglichkeit haben, den Hash-Wert von Eingabedateien zu berechnen. Dafür stehen dem Benutzer nachfolgende Optionen bereit:

- Auswahl an vordefinierten Hash-Funktionen
- Auswahl an Dateien, die zuvor in den Browser geladen werden

Auch hier muss der Benutzer die Auswahl explizit ausführen und erhält eine Anzeige des Kommandos im Befehlszeilenformat. Außerdem erfolgt die Ausgabe des berechneten Hash-Wertes.

Interaktion mit Dateien

Damit der Benutzer mit den zuvor definierten Anforderungen interagieren kann, soll es die Möglichkeit geben, für die gesamte Web-Applikation Dateien bereitzustellen. Dafür soll dem Benutzer Folgendes bereitstehen:

- lokale Dateien in den Web-Browser laden
- eine Übersicht aller Dateien
- einzelne Dateien löschen
- einzelne Dateien wieder ins lokale Dateisystem laden

Kommandozeile

Als letzte Anforderung der Web-Applikation soll der Benutzer alternativ mit einer Kommandozeile interagieren können. Das hat zum Ziel, dass Kommandos, welche nicht durch eine grafische Benutzeroberfläche umgesetzt sind, andernfalls über die Kommandozeile ausgeführt werden sollen. Benötigt man bspw. einen Zufallswert für die Eingabe eines Initialisierungsvektor in der GUI, so lässt sich das `rand`-Kommando parallel auf der Kommandozeile ausführen. Die Anforderungen werden durch die typische Verhaltensweise einer Kommandozeile definiert:

- Befehlszeile für die Eingabe eines Kommandos
- Anzeigebereich für die Ausgabe

Damit soll dem Benutzer ein vergleichbares Nutzungserlebnis geboten werden, wie es charakteristisch ist für ein Kommandozeilenprogramm.

3.2.2. Nicht-funktionale Anforderungen

Durch die nicht-funktionalen Anforderungen werden die Qualitätsansprüche an die Web-Applikation gestellt. Damit werden Eigenschaften abgedeckt, die zuvor nicht weiter konkretisiert wurden. [2, S.4]

Benutzbarkeit, Aussehen, Layout

Die Benutzeroberfläche der Web-Applikation soll möglichst funktional und intuitiv gestaltet werden, sodass dem Benutzer der Aufbau und die Auswahloptionen eines einzelnen OpenSSL-Kommandos näher gebracht wird. Der Aufbau sollte dabei klar strukturiert sein und eine konsistente Optik aufweisen für eine einheitliche Bedienung. Das Layout der Web-Applikation sollte einem responsive Web-Design entsprechen, um die Benutzung auch mit einem mobilen Gerät zu unterstützen.

Erweiterbarkeit

Durch eine komponentenbasierte Architektur der Web-Applikation soll die Erweiterbarkeit gewährleistet werden. Die Einbettung neuer Komponenten von OpenSSL-Kommandos als grafische Benutzeroberfläche soll unkompliziert umsetzbar sein. Infolgedessen soll sich der Funktionsumfang der Web-Applikation recht einfach erweitern lassen.

Zuverlässigkeit

Die Zuverlässigkeit bezüglich der Interpretation aller Auswahloptionen für die einzelnen OpenSSL-Kommandos ist eine Grundvoraussetzung. Wird ein Kommando fälschlicherweise aufgebaut, wäre das erzeugte Befehlszeilenformat des Kommandos zum einen irreführend für den Benutzer, zum anderen erzeugt solch ein Kommando eine unerwartete Ausgabe.

3.3. Aufbau und Funktionsweise

Die nachstehende Abbildung 9 zeigt den Aufbau der zu entwickelnden Web-Applikation. Diese stellt die Kombination aus den beiden zuvor beschriebenen Szenarien dar, unter der Einhaltung der funktionalen Anforderungen.

Insgesamt lässt sich die Web-Applikation zur Interaktion mit OpenSSL in zwei Hauptkomponenten unterteilen. Die erste ist die grafische Oberfläche (**A**), die sich als ein Tab-Container nochmals in vier weitere Tab-Komponenten unterteilt. Zur Auswahl stehen die Tab-Komponenten in der Navigationsleiste (**B**). Darunter fallen die Benutzerschnittstellen der einzelnen OpenSSL-Kommandos aber auch zusätzlich ein eigener Tab zur Verwaltung aller Dateien für die Web-Applikation.

Als Beispiel für eine Tab-Komponente ist die Auswahl des Hashes-Tabs zur Verwendung von Hash-Funktionen in Abbildung 9 zu sehen. Der Aufbau ist hier recht simpel und besteht lediglich aus zwei Input-Feldern (**C**, **D**). Zum einen soll die jeweilige Hash-Funktion für das Kommando `dgst` ausgewählt werden, zum anderen die zu verwendende Datei, welche zuvor im Files-Tab in den Browser geladen wurde. Bestätigt wird die

Auswahl mit dem Button (**E**) und führt das Kommando aus. Generell ist der Aufbau für die anderen Tab-Komponenten sehr ähnlich. Vor allem die Komponenten für die OpenSSL-Kommandos bestehen ausschließlich aus Input-Feldern, die repräsentativ für bestimmte Optionen und Argumente stehen.

Die zweite Hauptkomponente (**F**) ist die Nachbildung einer Kommandozeile. Diese besteht aus einem Input-Feld (**G**) und einer Textanzeige für die Ausgabe des Kommandos. Damit ist es möglich textuell mit OpenSSL zu interagieren. Die einzige funktionale Abhängigkeit zur grafischen Hauptkomponente besteht darin, dass in der grafischen Komponente **A** vorab die Dateien für das Kommando in den Browser geladen werden müssen. Wie in Abbildung 9 zu sehen ist, besteht dann die Möglichkeit über den Dateinamen das jeweilige Kommando auszuführen.

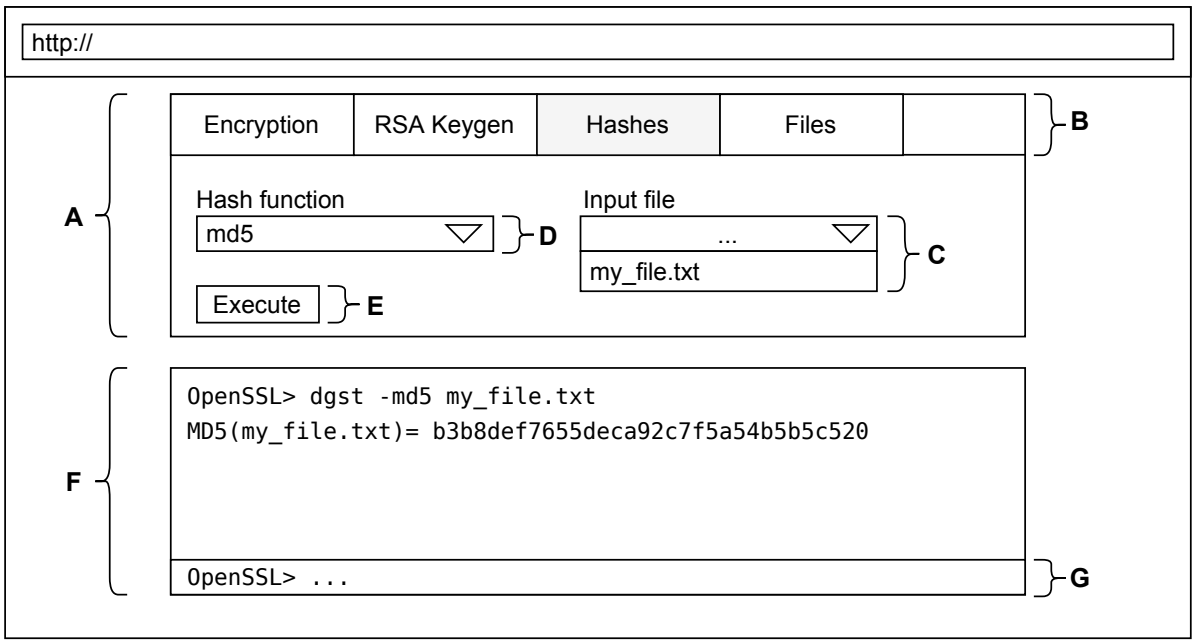

Abbildung 9: Entwurf der Web-Applikation

3.4. Technische Analyse

In der technischen Analyse geht es darum, Software bzw. Technologien, welche für die Implementierung der Web-Applikation in Frage kommen, gegeneinander abzuwägen. Das betrifft einerseits die Kompilierung von OpenSSL zu WebAssembly und andererseits die Entwicklung des Front-Ends.

3.4.1. WebAssembly C/C++-Toolchains

OpenSSL ist hauptsächlich in der Sprache C[7] geschrieben und verwendet in Bezug auf das Kommandozeilenprogramm die C-Standardbibliothek, welche sich unter anderem um Systemaufrufe kümmert. Damit das OpenSSL-Kommandozeilenprogramm in einer Browserumgebung ausführbar ist, muss gewährleistet sein, dass diese Systemaufrufe emuliert werden. Allein die Kompilierung von OpenSSL zu WebAssembly ohne die Implementierung von Schnittstellen für Systemaufrufe, wäre für die Ausführung im Browser nicht nützlich.

Daher wird eine WebAssembly C/C++-Toolchain benötigt, die die Systemschnittstellen aus den Standardbibliotheken implementiert, damit das WebAssembly-Modul bspw. im Fall einer Datei-Operation weiß, wie diese im Browser behandelt werden muss. Zur Auswahl stehen dafür Emscripten oder eine WASI-kompatible Toolchain. Zur WASI-Toolchain zählen Wasienv[8] oder WASI SDK[9].

Emscripten

Die Emscripten-Toolchain wurde von Anfang an mit dem Ziel konzipiert, dass eine Ausführung von C/C++-Programmen im Browser möglich ist (siehe Kapitel 2.2.2). Das ist vor allem durch die Eigenimplementierung von Funktionen aus der C-Standardbibliothek realisiert worden. [23] Um mit diesen APIs im Browser zu kommunizieren, entsteht bei der Kompilierung durch Emscripten ein sogenannter JS-Glue-Code. Dieser fungiert als eine Schnittstelle für die Wasm-Runtime im Browser, die die Interaktion zwischen JavaScript und dem WebAssembly-Modul erlaubt und verschiedene Funktionen wie ein virtuelles Dateisystem bereithält. Zudem implementiert Emscripten nicht nur Funktionen aus der C-Standardbibliothek, sondern auch weitere Schnittstellen zur Verwendung von C/C++-Bibliotheken wie Simple DirectMedia Layer (SDL) oder WebGL. [7]

Das bringt insgesamt viele Vorteile mit sich, wenn das existierende C/C++-Programm die oben genannten Bibliotheken verwendet und der Browser als Host für die Laufzeitumgebung des Wasm-Moduls gewählt wird. Zur Interaktion mit dem Wasm-Modul muss dafür lediglich der Glue-Code in die Web-Applikation integriert werden.

Der Nachteil an Emscripten ist jedoch, dass die Schnittstellenimplementierungen der C/C++-Bibliotheken nicht standardisiert sind. [23] Das heißt, die Verwendung des OpenSSL Wasm-Moduls ist nur in Kombination mit dem zugehörigen Emscripten Glue-Code möglich.

[7] `https://github.com/openssl/openssl` (siehe Languages-Statistik)
[8] `https://github.com/wasienv/wasienv`
[9] `https://github.com/WebAssembly/wasi-sdk`

WASI-kompatible Toolchain

WebAssembly ist eine Assemblersprache für eine virtuelle Maschine. Aus diesem Grund wird bei der Kompilierung zu Wasm auch keine Zielplattform festgelegt, da Wasm primär zur Ausführung im Web entworfen wurde. [11, S.1] Mit dem Ziel WebAssembly allgemein auch außerhalb einer Browserumgebung auszuführen, benötigt Wasm einerseits eine Laufzeitumgebung, aber auch Schnittstellen für ein virtuelles Betriebssystem. Solche Schnittstellen sollen durch das WebAssembly System Interface (WASI) standardisiert werden, um systemorientierte APIs für WebAssembly bereitzustellen. Mit WASI *libc* soll es außerdem eine vollständige Unterstützung der C-Standardbibliothek geben, die bei der Kompilierung von C/C++-Programmen zu WebAssembly mit den bereits genannten WASI-kompatible Toolchains berücksichtigt wird. [25]

Zwar ist die Ausführung von WebAssembly außerhalb des Browsers nicht Teil dieser Arbeit und wird auch nicht näher erläutert, jedoch gibt es die Möglichkeit, WASI-Module gezielt über ein Polyfill im Web-Browser auszuführen. Die Aufgabe eines Polyfills ist die Bereitstellung von Technologien, die nicht nativ vom Web-Browser unterstützt werden. [27] Ein Beispiel für WASI im Browser ist Wasmer-JS[10]

Im Unterschied zur Emscripten-Toolchain entsteht bei der Erzeugung eines WASI-Moduls kein zusätzlicher JS-Glue-Code. Der Nachteil ist dabei, dass die Verwendung des WASI-Moduls im Browser von zusätzlichen JS-Bibliotheken abhängig ist, die das notwendige Polyfill implementieren. Das bedeutet in Bezug auf die Kompilierung von OpenSSL zu WebAssembly, dass die Verwendung im Browser nicht nativ möglich ist. Hingegen ist der erzeugte Glue-Code der Emscripten-Toolchain reines JavaScript, welches ohne weitere Abhängigkeiten zur Interaktion mit dem Wasm-Modul verwendet werden kann.

Abschließend muss ergänzt werden, dass WASI noch in der Entwicklung steht, und somit hinsichtlich der Zuverlässigkeit für die Web-Applikation eher ungeeignet ist.

3.4.2. Entwicklung des Front-Ends

Die Entwicklung des Front-Ends ist grundsätzlich nicht davon abhängig, wie OpenSSL zu WebAssembly kompiliert wird. Die einzige Voraussetzung, die aufgrund der Interaktion mit dem Wasm-Modul an die Web-Applikation gestellt wird, ist die Verwendung von JavaScript. Da JS ohnehin zur dynamischen Gestaltung der Benutzeroberfläche verwendet werden soll, stellt dies kein Problem für die Web-Applikation dar.

Des Weiteren ist das Ziel, die Web-Applikation ausschließlich Client-seitig auszuführen. Damit inbegriffen ist zum einen die Ausführung einzelner OpenSSL-Kommandos, aber

[10] `https://github.com/wasmerio/wasmer-js`

auch die Navigation in der Anwendung selbst. Ein Ansatz dafür sind SPAs. Hier wird die Anwendung im Vorfeld durch ein Front-End-Framework generiert und statisch durch einen Web-Server an den Client ausgeliefert. Der Vorteil hierbei ist, dass die Anwendung dynamisch mit JS im Browser aufgebaut wird und sich dadurch sehr ähnlich wie eine Desktopanwendung verhält. Im Unterschied zum klassischen Aufbau einer Webseite, wo mehrere HTML-Dokumente miteinander verlinkt werden, entfällt dementsprechend die Aktualisierung einzelner HTML-Dokumente durch den Browser. Die Navigation auf der Webseite wird intern durch JS verarbeitet und parallel dazu das DOM entsprechend manipuliert.

Das Verhalten einer SPA ist relativ umständlich mit reinem HTML, CSS und JS umzusetzen. Solch ein Verhalten ist jedoch auch nicht zwingend notwendig für die Realisierung der Web-Applikation. Prinzipiell lässt sich die Anwendung auch mit reinem HTML, CSS und JS umsetzen. Problematisch wird es erst, wenn man größer skaliert und das Projekt dadurch schnell unübersichtlich werden kann. Das bezieht sich unter Anbetracht der nicht-funktionalen Anforderungen (Kap. 3.2.2) darauf, dass nachträglich auch weitere Kommandos als GUI umgesetzt werden sollen.

Insgesamt sprechen die Anforderungen und das Ziel der Web-Applikation für die Umsetzung als SPA. Ein Front-End-Framework wie React bietet von Grund auf einen Baukasten, womit sich das Projekt strukturieren lässt, aber vor allem die Entwicklung vereinfacht.

4. Implementierung der Web-Applikation

In diesem Kapitel wird die Vorgehensweise zur Umsetzung der Web-Applikation näher erläutert. Dafür werden zunächst die Entwicklungsumgebung und die zu verwendenden Technologien dargelegt.

Als Plattform für die Entwicklung wird die Linux-Distribution Ubuntu 20.04 gewählt. Diese bringt gleichzeitig die Bash-Shell mit, welche später relevant für ein automatisiertes Build-Skript ist. Das Kompilieren von OpenSSL zu WebAssembly erfolgt mit der Emscripten-Toolchain. Hierzu muss erst das Emscripten Software Development Kit (SDK)[11] auf dem System installiert werden. Nach einer erfolgreichen Installation kann die Emscripten-Umgebung mit folgendem Befehl initialisiert werden:

```
$ source /path/to/emscripten_dir/emsdk_env.sh
```

Für die Entwicklung des Front-Ends wird eine Node.js[12]-Umgebung eingerichtet mit dem Paketmanager npm[13]. Darüber lassen sich alle benötigten Pakete für die Web-Applikation verwalten. Unter anderem auch die Create-React-App[14]-Umgebung für das Initialisieren einer React-Anwendung.

Das zusammen bildet die Grundlage für die Entwicklungsumgebung, welche im Folgenden genutzt wird, um die Web-Applikation zu implementieren.

4.1. Kompilierung von OpenSSL zu WebAssembly

Die Vorgehensweise bei der Kompilierung von OpenSSL zu WebAssembly mit Emscripten ist sehr ähnlich zum nativen Kompilierungsvorgang mit der GNU-Toolchain. OpenSSL verwendet dafür ein benutzerdefiniertes Build-System, wo vorab über ein Konfigurationsskript Optionen für die einzelnen Bibliotheken festgelegt werden können. Daraufhin erstellt das Konfigurationsskript das notwendige Makefile, um mit Hilfe des Source-Codes die Bibliotheken zu bauen. [6]

Um nun das OpenSSL-Projekt mit der Emscripten-Toolchain zu kompilieren, sind keine großen Änderungen notwendig. Die Vorgehensweise bleibt gleich, jedoch werden statt den Standardwerkzeugen aus dem GNU-Projekt speziell durch Emscripten angepasste Skripte verwendet, um bspw. das Konfigurationsskript oder das Makefile auszuführen.

[11] https://emscripten.org/docs/getting_started/downloads.html#sdk-download-and-install
[12] https://nodejs.org/en/
[13] https://www.npmjs.com/
[14] https://reactjs.org/docs/create-a-new-react-app.html

[4] Der C-Compiler `gcc` wird außerdem durch das Emscripten Compiler-Front-End `emcc` ersetzt.

Für die weitere Entwicklung der Web-Applikation wird der Kompilierungsvorgang mittels eines Build-Skripts automatisiert, um das ausführbare Wasm-Programm zu erzeugen. Im weiteren Verlauf werden die relevanten Stellen im Skript näher erläutert.

```
export CC=emcc
export CXX=emcc

export LDFLAGS="\
  -s ENVIRONMENT='web'\
  -s FILESYSTEM=1\
  -s MODULARIZE=1\
  -s EXPORT_NAME=OpenSSL\
  -s EXTRA_EXPORTED_RUNTIME_METHODS=\"['callMain', 'FS']\"\
  -s INVOKE_RUN=0\
  -s EXIT_RUNTIME=1\
  -s EXPORT_ES6=1\
  -s USE_ES6_IMPORT_META=0"
```

Listing 6: Anpassung der Umgebungsvariablen

Listing 6 zeigt die verwendeten Umgebungsvariablen, die von dem konfigurierten Makefile genutzt werden sollen. Mit den Variablen `CC` und `CXX` wird der zuständige C- bzw. C++-Compiler gesetzt. Damit bei der Kompilierung das Emscripten Compiler-Front-End verwendet wird, werden diese Variablen durch `emcc` ersetzt. Das hat zur Folge, dass hinterher bei der Ausführung des Makefiles `emcc` statt dem standardmäßigen C-Compiler `gcc` genutzt wird.

Von Bedeutung sind vor allem die Link-Optionen in `LDFLAGS`, die zur Link-Time verwendet werden, um einen angepassten JS-Glue-Code zu erzeugen. Emscripten stellt dafür eine Liste[15] an Optionen zur Verfügung, die letztlich an das Compiler-Front-End `emcc` übergeben werden.

Für den Anwendungsfall der Web-Applikation sind alle verwendeten Optionen im Listing 6 zu sehen. Im Folgenden werden die relevanten Optionen näher beschrieben:

`ENVIRONMENT` Mit dieser Option wird die Laufzeitumgebung festgelegt. Da die Ausführung der Anwendung im Web-Browser erfolgen soll, wird hier explizit das Web ausgewählt.

[15] `https://github.com/emscripten-core/emscripten/blob/main/src/settings.js`

`FILESYSTEM` Hiermit wird die Unterstützung des virtuellen Dateisystems aktiviert. Daraufhin ist es möglich, über die File System API von Emscripten dateibasierte Operationen durchzuführen.

`INVOKE_RUN` Diese Option trägt dazu bei, dass das Wasm-Modul für jedes Kommando individuell initialisiert werden kann. Denn damit wird festgelegt, dass die `main()`-Funktion programmatisch zu einer bestimmten Zeit aufgerufen wird.

`EXIT_RUNTIME` Hiermit wird bestimmt, dass die Runtime nach der Ausführung nicht beendet wird. Das hat zur Folge, dass sich Kommandos wiederholt ausführen lassen.

`EXTRA_EXPORTED_RUNTIME_METHODS` Die hier angegebenen Methoden werden im Glue-Code implementiert und können im Nachhinein in JS verwendet werden, um mit dem Wasm-Modul zu interagieren. Im Zusammenhang mit OpenSSL werden `main()` und das Dateisystem-Objekt `FS` exportiert.

Die restlichen Optionen dienen zum Importieren des JS-Glue-Codes in die Applikation unter Verwendung von neueren JavaScript-Features.

```
15 emconfigure ./Configure \
16   no-hw \
17   no-shared \
18   no-asm \
19   no-threads \
20   no-dso \
21   linux-x32 \
22   -static\
```

Listing 7: Verwendung von emconfigure

Listing 7 zeigt eine an WebAssembly angepasste Konfiguration der OpenSSL-Bibliotheken. Der Aufruf durch `emconfigure` unterscheidet sich grundsätzlich kaum zur standardmäßigen Vorgehensweise. Hierfür wird das eigentliche Konfigurationsskript mit den zusätzlichen Optionen als Argument an `emconfigure` übergeben.

Primär wird bei der Konfiguration berücksichtigt (`no-shared, no-dso, -static`), dass zur Kompilierung des Wasm-Excecutable nur statische Bibliotheken verwendet werden. [4] Des Weiteren werden einzelne Features deaktiviert wie Assembler-Routinen (`no-asm`), Multithreading (`no-threads`) und Hardware-Support (`no-hw`), die grundsätzlich nicht von WebAssembly unterstützt werden. Als Plattform wird explizit eine x32-

Architektur angegeben, da WebAssembly den Speicher nur über 32-bit breite Zeiger adressiert. [19]

```
25 sed -i 's/$(CROSS_COMPILE)//' Makefile
26
27 emmake make
28 mv apps/openssl apps/openssl.js
```

Listing 8: Verwendung von emmake

Grundlage für die Ausführung des Makefiles ist die spezielle Anpassung des Konfigurationsskripts durch `emconfigure`. Zuvor muss das Makefile jedoch leicht modifiziert werden, da das Konfigurationsskript den Emscripten-Compiler richtigerweise als Cross-Compiler erkennt, jedoch damit die spezifischen Emscripten-Pfade verfälscht. Emscripten selbst wird nicht als natives Build-System erkannt, wodurch die tatsächlichen Installationspfade zu dem Emscripten-Compiler durch die vorangestellte `CROSS_COMPILE`-Konstante falsch sind. Mit dem `sed`-Kommando in Listing 8 wird die Konstante vor den jeweiligen Pfaden entfernt.

Daraufhin kann das Makefile mit der üblichen Vorgehensweise aufgerufen werden. In Listing 8 wird dafür das `make`-Kommando als Argument an das Skript `emmake` übergegeben und anschließend ausgeführt.

Nach der Kompilierung entsteht im *apps*-Verzeichnis des Source-Codes das Wasm-Executable *openssl.wasm* und eine nicht näher spezifizierte Datei *openssl*. Darin befindet sich der JS-Glue-Code, weshalb diese am Ende von Listing 8 zu einer JS-Datei umbenannt wird.

Abschließend muss erwähnt werden, dass das Build-Skript von keiner spezifischen OpenSSLVersion abhängt. Erfolgreich ausgeführt wurde das Build-Skript mit den OpenSSL-Versionen 1.1.1k und der 15. Alpha-Version 3.0.

Grundlage dafür ist die offizielle Dokumentation im GitHub-Repository[16].

4.2. Integration des Wasm-Binärcodes

Das in 4.1 erzeugte Wasm-Modul mit dem zugehörigen JS-Glue-Code bildet den Kern der Web-Applikation. Damit ist es nun möglich, das Wasm-Modul in der Anwendung zu instanziieren und über die Runtime-Methoden OpenSSL-Kommandos auszuführen.

[16] `https://github.com/openssl/openssl/blob/master/INSTALL.md#building-openssl`

Für die Interaktion mit den Runtime-Methoden wird zusätzlich eine global verfügbare Klasse namens `Command` zur Verfügung gestellt, die sowohl den Aufruf der Runtime-Methoden verwaltet als auch eine Reihe von Methoden für die Verarbeitung der einzelnen Kommandos implementiert.

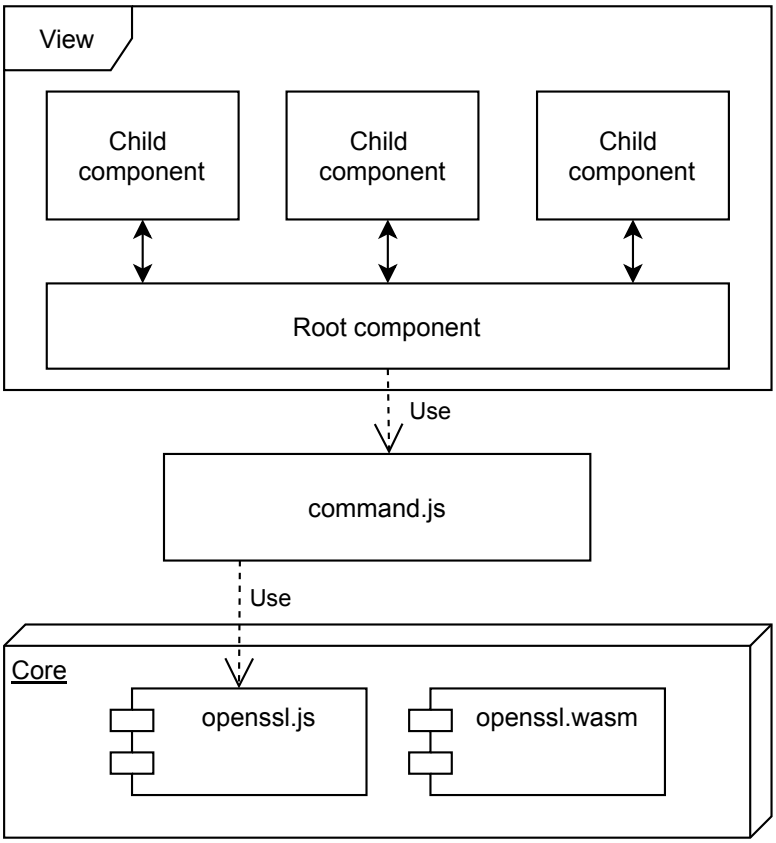

Abbildung 10: Implementierung der Kernkomponenten in die Web-Applikation

Der Grund für die gewählte Implementierung in Abbildung 10 hängt damit zusammen, dass der Aufruf der Runtime-Methoden in `openssl.js` strikt von der View-Logik getrennt werden soll. Dafür implementiert die Singleton-Klasse `Command` eine Wrapper-Methode, wo das ausgewertete Kommando als String mit den zusätzlichen Dateien als File-Objekte übergeben wird. In dieser Methode erfolgt anschließend der Aufruf der Runtime-Methoden. Das macht die Klasse `Command` zur einzigen Schnittstelle zwischen der View-Logik und den Kernkomponenten.

Die einzelnen Bestandteile der `Command`-Klasse lassen sich wie folgt beschreiben:

Import des JS-Glue-Code Grundvoraussetzung für die Klasse ist der Glue-Code des zugehörigen Wasm-Moduls. Dieser lässt sich durch ein Import-Statement importieren, wie in Listing 9 zu sehen ist.

```
1 import OpenSSL from './openssl';
```

Listing 9: Import des Emscripten Glue-Codes

Anschließend steht der Import `OpenSSL` als eine Funktion bereit, die ein Promise-Objekt zurückgibt. Das ist beabsichtigt durch die angegebene Option `-s MODULARIZE=1` beim Kompilierungsvorgang. Die Aufgabe des Promise-Objekts besteht darin, dass das Wasm-Modul nach erfolgreicher Instanziierung innerhalb eines Callback verwendet werden kann.

Klassenkonstruktor Das Standardverhalten der Emscripten-Runtime sieht vor, dass der Wasm-Binärcode mit jeder Instanziierung erneut über die `fetch()`-Funktion abgerufen wird. Das würde für die Verwendung der Web-Applikation bedeuten, dass mit jedem ausgeführten Kommando der Wasm-Binärcode erneut heruntergeladen werden muss. Um das Datenaufkommen zu reduzieren, wird der Binärcode im Konstruktor der Klasse abgerufen und später dem Glue-Code zur Instanziierung übergeben. Da die Klasse ein Singleton ist, erfolgt der Abruf des Wasm-Binärcodes somit nur einmalig.

Methode zur Kommandoausführung Diese Methode in Listing 10 stellt den wichtigsten Teil der Klasse `Command` dar und dient zur Ausführung der Kommandos, die über die Benutzeroberfläche der Anwendung aufgerufen werden. Nachfolgend werden einzelne Abschnitte der Methode näher erläutert.

```
1 /**
2 * @param {string} args OpenSSL Command
3 * @param {File[]} files Command files to process
4 * @param {string} text Command text to process
5 */
6
7 async run(args, files = null, text = '')
```

Listing 10: Parameter der Methode `run`

Die Methode nimmt insgesamt drei Parameter entgegen, wobei `files` und `text` optional sind. Das Kommando steht zunächst als String in dem Parameter `args` und wird vor der Übergabe an `callMain()` in ein String-Array umgewandelt. Die optionalen Parameter dagegen dienen als Daten, die vom virtuellen Dateisystem verarbeitet werden. Im Fall, dass bspw. ein Text zur Verschlüsselung übergeben wird, muss der String zunächst in eine Datei umgewandelt werden. Anschließend wird diese Datei an das Dateisystem übergeben und der Inhalt durch OpenSSL verschlüsselt.

Listing 11 zeigt, wie die Ausführung des Wasm-Moduls über das Module-Objekt[17] in JavaScript kontrolliert werden kann. Das heißt, dieses Objekt enthält alternative Implementierungen der Runtime-Methoden, die vor der Ausführung an den Glue-Code übergeben werden. In Bezug auf die Web-Applikation betrifft das unter anderem die Instanziierung des Wasm-Moduls. Die Funktion `instantiateWasm` enthält das vorab kompilierte Wasm-Modul, welches daraufhin im Glue-Code instanziiert wird.

```
const moduleObj = {
  thisProgram: 'openssl',
  instantiateWasm: function (imports, successCallback) {
    wasmModule.then((module) => {
      WebAssembly.instantiate(module, imports).then(successCallback);
    });
    return {};
  },
  print: function (line) {
    output.stdout += line + '\n';
  },
  printErr: function (line) {
    output.stderr += line + '\n';
  },
};
```

Listing 11: Emscripten Module-Object

Des Weiteren werden die Funktionen `print` und `printErr` überschrieben, die im Normalfall `stdout` bzw. `stderr` aus dem Kommandozeilenprogramm in der Browser-Konsole ausgeben würden. Da die Web-Applikation beide Ausgaben für die Anzeige in der Benutzeroberfläche verwendet, werden diese Funktionen dahingehend abgeändert, dass die Ausgaben in einem Objekt gespeichert werden.

Das Code-Listing 12 zeigt den gekürzten Aufruf des Wasm-Moduls. Von Bedeutung sind vorrangig die Interaktionen mit den Runtime-Methoden `FS` und `callMain()`.

```
OpenSSL(moduleObj)
  .then((instance) => {
    // ...
    instance['FS'].writeFile(file.name, file.buffer);
    // ...
    instance.callMain(argsArray);
    // ...
```

[17] `https://emscripten.org/docs/api_reference/module.html`

```
8   })
9   .catch((error) => (output.stderr = `${error.name}: ${error.message}`))
10  .finally(() => this.resultSubject.next(output));
```

Listing 12: Aufruf des WebAssembly-Moduls

Als Parameter wird der Funktion das zuvor beschriebene Module-Objekt übergeben. Um den Rückgabewert der asynchronen Operation des Promise-Objekts zu erhalten, wird ein Callback mit `then()` aufgerufen. Der Rückgabewert ist in dem Fall das erfolgreich instanziierte Wasm-Modul. Damit lassen sich alle zuvor exportierten Runtime-Methoden verwenden. Das Objekt `FS` besitzt eine große Auswahl[18] von Operationen, um mit dem virtuellen Dateisystem zu interagieren. Wichtig ist an dieser Stelle, dass die Datei vor dem Aufruf von `callMain()` im Dateisystem zur Verfügung steht. Erst dann kann die Datei von dem jeweiligen Kommando gelesen werden.

Output-Objekt Wie bereits erwähnt, werden die Ausgaben von `stdout` und `stderr` in einem einzigen Objekt abgespeichert. Ebenfalls wird die Datei, welche durch das Kommando erzeugt wurde, in diesem Objekt abgelegt. Damit die View-Logik die Daten entsprechend anzeigen kann, wird das Objekt mittels eines Oberservable an die Oberserver in der View-Logik verschickt. Der Grund dafür ist die asynchrone Ausführung des Wasm-Moduls, da die Ausgabe zu keinem bestimmten Zeitpunkt erwartet werden kann. Daher werden die Observer mit dem Output-Objekt benachrichtigt, sobald die Ausführung beendet wurde (siehe Code-Listing 12, Z.10).

4.3. Implementierung des Front-Ends

Das interaktive Front-End der Web-Applikation soll mit Hilfe des JS-Framework React realisiert werden. Dafür wird die Anwendung, unter Voraussetzung einer installierten npm-Paketverwaltung, wie folgt in der Kommandozeile durch die Create-React-App-Umgebung erstellt:

```
$ npx create-react-app openssl-webapp
```

Anschließend entsteht ein Projektverzeichnis namens *openssl-webapp*[19] mit den grundlegenden Projektdateien einer React-Anwendung. Darunter befindet sich unter anderem die Komponente `App`, welche fortan die oberste Ebene der Komponenten-Hierarchie darstellt und weitere Komponenten zu einer Benutzeroberfläche bündelt.

[18] `https://emscripten.org/docs/api_reference/Filesystem-API.html`

[19] Das gesamte Projektverzeichnis inklusive des Quellcodes der Web-Applikation befindet sich auf der CD im Anhang

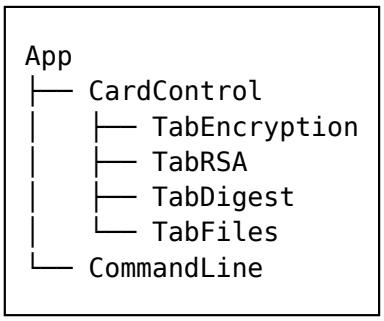

Abbildung 11

Grundlage für die Gestaltung des hierarchischen Aufbaus (Abb. 11) ist der Entwurf (Kap. 3.3) der Web-Applikation. Die daraus resultierenden Komponenten besitzen einen eigenen Aufgabenbereich, welcher durch die funktionalen Anforderungen (Kap. 3.2.1) näher beschrieben wird. Das Ziel ist dabei, das jeweilige OpenSSL-Kommando über UI-Elemente zu interpretieren und über die `Command`-Klasse an das Wasm-Modul zu übergeben.

Aufbau der Komponenten

Für die Umsetzung einer solchen Hierarchie werden zunächst die einzelnen Komponenten im Projektverzeichnis erstellt. Die Strukturierung im Verzeichnis selbst orientiert sich dabei an der Abbildung 11.

Eine typische React-Komponente besteht aus folgenden Dateien:

- Component.jsx
- Component.css
- Component.test.js

An dieser Stelle fällt auf, dass ein HTML-Dokument kein Bestandteil einer Komponente ist. Grund dafür ist die Verwendung von JavaScript XML (JSX). Als eine syntaktische Erweiterung zu JavaScript kann damit eine HTML-ähnliche Syntax in JS verwendet werden. Exemplarisch für den Aufbau von JSX dient ein Auszug der App-Komponente in Listing 13.

```
1  import CardControl from '../card-control/CardControl';
2  import CommandLine from '../../components/command-line/CommandLine';
3  import './App.css';
4  // ...
5  function App() {
6  // ...
7    return (
8      <div className="App">
9        <CardControl runCommand={...}>
10       </CardControl>
11       <CommandLine runCommand={...} result={...}>
12       </CommandLine>
```

```
13      </div>
14    );
15  }
16
17  export default App;
```

Listing 13: Auszug der App-Komponente

Hierbei wird deutlich, dass sich die Anzeige einer Komponente über die Rückgabe der JS-Funktion definiert. Das CSS-Styling erfolgt durch den Import des entsprechenden Stylesheets und kann, wie auch in HTML, in den JSX-Tags verwendet werden.

Die Rückgabe der JS-Funktion kann außerdem auf weitere Komponenten verweisen. Damit wird eine ähnliche Abstraktion erreicht, wie die Verwendung eines gewöhnlichen HTML-Elements, um jede Detailstufe der Web-Applikation durch eine Komponente auszudrücken. Wie sich `CardControl` bzw. `CommandLine` im Code-Listing 13 zusammensetzt, ist für die `App`-Komponente an dieser Stelle unbekannt. Zur Kommunikation bieten die Komponenten eine Schnittstelle mittels sogenannter Props, um den Austausch von Daten und Funktionen zu erlauben. So wird bspw. die Methode zur Ausführung eines Kommandos über das Prop `runCommand` an die Child-Komponenten von `App` übergeben.

Darstellung der Web-Applikation

Für die Darstellung und Manipulation des DOM im Web-Browser ist die `render()`-Funktion des ReactDOM-Objekts zuständig.

```
1  ReactDOM.render(
2    <App />,
3    document.getElementById('root')
4  );
```

Listing 14: Rendering der App-Komponente

Die Funktion in Listing 14 nimmt insgesamt zwei Argumente entgegen. Ersteres ist die React-Komponente, welche im Browser gerendert werden soll. Das umfasst auch alle Child-Komponenten. Des Weiteren wird ein DOM-Knoten angegeben, der als Container für die angegebene Komponente dient.

Wird ein Production-Build der Web-Applikation erzeugt, besteht die eigentliche Anwendung nur aus JS- und CSS-Dateien. Für die Darstellung im Web-Browser wird zusätzlich eine statische `index.html` ausgeliefert, die einen einzigen DOM-Knoten mit `id="root"` besitzt.

Für eine einheitliche und moderne Oberflächengestaltung wird das CSS-Framework Bootstrap[20] gewählt. Das Framework stellt eine Reihe von CSS-Klassen zur Verfügung, die sich um die Gestaltung der HTML-Elemente kümmern. Darüber hinaus bietet Bootstrap ein Grid-System, das sich an der Bildschirmauflösung orientiert, um ein responsive Web-Design zu realisieren. Die HTML-Elemente können in Reihen und Spalten eingeteilt werden, sodass sich je nach Bildschirmauflösung die Anordnung der Elemente dynamisch ändert.

Abbildung 12 zeigt die Implementierung der Web-Applikation mit UI-Elementen aus dem standardmäßigem Bootstrap-Theme. Der grundlegende Aufbau wird in Listing 13 durch die Komponenten `CardControl` und `CommandLine` festgelegt. Dadurch unterteilt sich die Anwendung in einen GUI- und CLI-Bereich. `CardControl` beinhaltet zusätzlich die einzelnen Tab-Komponenten gemäß der Komponenten-Hierarchie (Abb. 11). Über die Auswahl in der Navigationsleiste wird der Anzeigebereich in `CardControl` mit dem jeweiligen Tab-Inhalt dynamisch aktualisiert. In Abbildung 12 ist die Auswahl der `TabEncryption`-Komponente zu sehen. Hier wird eine ausgewählte Datei aus dem Input-Dropdown mit AES-256 im CBC-Modus verschlüsselt und der Output wird Base64-codiert (Option -a) auf der Kommandozeile angezeigt. Dabei wird das Passwort als String mitgegeben, ebenso wie der Initialisierungsvektor. Als Schlüsselableitungsfunktion ist PBKDF2 ausgewählt.

Das Layout innerhalb eines Tabs richtet sich nach dem Grid-System aus Bootstrap. Dadurch werden die HTML-Elemente strukturiert angeordnet. Die Interaktion mit den einzelnen HTML-Elementen gestaltet sich ebenfalls dynamisch. Wird bspw. in Abbildung 12 als *Input* die *Text*-Option ausgewählt, ändert sich das darunterliegende Dropdown-Feld zu einer Texteingabe. Dieses Verhalten sorgt dafür, dass die GUI kompakt bleibt und nur die notwendigen Eingabefelder für das Kommando angezeigt werden.

Weitere Darstellungen zu den Tab-Komponenten wie `TabGenrsa` (Abb. 15), `TabDigest` (Abb. 16) und `TabFiles` (Abb. 17) befinden sind im Anhang A.

[20] `https://getbootstrap.com/`

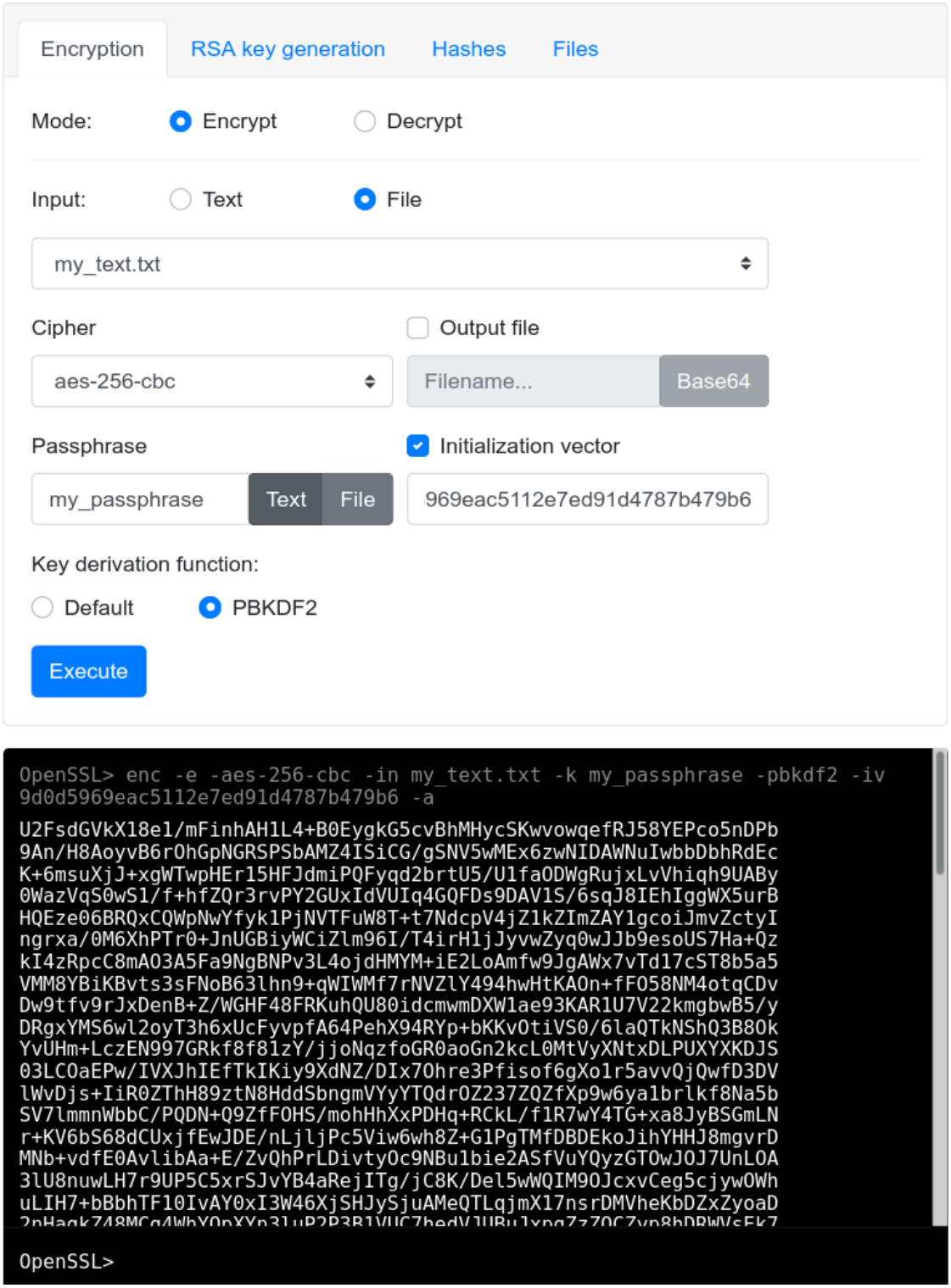

Abbildung 12: Darstellung der Web-Applikation

State einer Komponente

Die OpenSSL-spezifischen Tabs in der CardControl-Komponente (siehe bspw. Abb. 14 in Anhang A) stehen jeweils für ein eigenes OpenSSL-Kommando. Da die anwählbaren Optionen durch die GUI vielseitig und vor allem individuell sind, bietet es sich an, die Tab-Komponenten als jeweils eigenen State zu verwalten. Dieser State repräsentiert die ausgewählten Optionen für ein Kommando.

Das React-Framework bietet dafür seit Version 16.8 spezielle Funktionen („Hooks“) an, um sich in die jeweiligen Lifecycles einer Komponente „einzuhaken“. [13] Um einer Komponente einen State zu geben, kommt primär die `useState`-Hook in der Web-Applikation zum Einsatz.

```
1 const [genrsa, setGenrsa] = useState({
2   outFile: 'my_rsa.pem',
3   numbits: '1024',
4 });
```

Listing 15: useState-Hook am Beispiel der TabGenrsa-Komponente

Listing 15 zeigt den Initialzustand der TabGenrsa-Komponente. Die `useState()`-Funktion gibt zwei State-Variablen zurück, die über ein Array-Destructuring definiert werden können. Das bedeutet, dass sich die Funktion in die Variablen `genrsa` und `setGenrsa` auflöst. Die erste Variable repräsentiert dabei den aktuellen State der Komponente. In diesem Fall, das Objekt mit `outFile:'my_rsa.pem'` und `numbits:'1024'`. Mit `setGenrsa` kann der State aktualisiert werden, um ein Re-Render der UI auszulösen. Daraufhin werden bspw. GUI-Elemente aktualisiert.

Im Beispiel der TabGenrsa-Komponente (Abb. 15 in Anhang A), besitzen die Input-Felder ein `onChange()`-Event. Damit ist das Input-Feld an eine Funktion gekoppelt, die bei jeder Input-Änderung ausgeführt wird. In Bezug auf das Feld *Output file*, wird somit bei einer Texteingabe automatisch der State von `outFile` in dem Objekt `genrsa` mit dem entsprechenden Text aktualisiert.

Auf diese Art werden die Input-Daten des Benutzers in einem State festgehalten. Entscheidet sich der Benutzer zur Ausführung der ausgewählten Optionen, durchläuft das Objekt `genrsa` eine Funktion, in der über die einzelnen Eigenschaften des Objekts iteriert wird. Daraus wird schließlich ein gültiges OpenSSL-Kommando als String erzeugt und schlussendlich durch die `Command`-Klasse ausgeführt.

Diese Implementierung zieht sich über alle OpenSSL-spezifischen Komponenten und ist die Grundlage für die Interpretation von Kommandos durch GUI-Elemente.

5. Evaluation

Die Grundlage für die Realisierung der Web-Applikation bietet das Kompilierungsziel WebAssembly im Zusammenspiel mit den Web-Technologien wie HTML, CSS und JS. Im Folgenden sollen die hauptsächlichen Ziele der Implementierung im Zusammenhang mit OpenSSL bewertet werden.

WebAssembly als Kompilierungsziel für OpenSSL

Die Kompilierung (Kap. 4.1) mit Hilfe der Emscripten-Toolchain zeigt, dass sich WebAssembly durchaus als Binärformat für existierende C-Programme eignet. Zu diesem Ergebnis führt eine Vorbereitung, die sich am Anfang dieser Arbeit als schwierig erwiesen hat. Dafür musste zunächst erarbeitet werden, aus welchen Bestandteilen sich das OpenSSL-Projekt zusammensetzt und wie man gezielt das Kommandozeilenprogramm kompiliert. Erste Versuche die Kompilierung mit Emscripten durchzuführen, bestanden darin, die Objektdateien des OpenSSL-Programms separat mit `emcc` zu verbinden. Die Ausführung des erzeugten Wasm-Moduls war jedoch fehlerhaft und somit unbrauchbar für die Web-Applikation. Alternativ hätte man die reinen algorithmischen Implementierungen von OpenSSL für ein eigenes C/C++-Kommandozeilenprogramm nutzen können, welches man wiederum zu WebAssembly kompiliert. Dieser Aufwand würde allerdings den Umfang dieser Bachelorarbeit sprengen.

Nachdem ersichtlich war, dass *emmake* mit der Ausführung des Makefiles im OpenSSL-Projekt auch direkt das Kommandozeilenprogramm als WebAssembly-Modul erzeugt, erwies sich die Vorgehensweise als recht unkompliziert. Diese unterscheidet sich kaum zur Standardvorgehensweise wie man sie von nativen Toolchains kennt. Die entscheidende Rolle für eine erfolgreiche Kompilierung des OpenSSL-Projekts spielt dabei Emscripten. Ohne die Eigenimplementierungen von Schnittstellen für C/C++-Bibliotheken, wären die Funktionalitäten des erzeugten WebAssembly-Binärcodes weniger umfangreich.

OpenSSL als grafische Benutzeroberfläche

Aufgrund der klaren Strukturierung eines OpenSSL-Kommandos, eignen sich diese sehr gut zur Umsetzung als eine grafische Benutzeroberfläche. Die offiziellen Dokumentationen über die einzelnen Kommandos bieten die Referenz, wie ein Kommando aufgebaut ist und welche Argumente bzw. Optionen erwartet werden. Damit sind die Eingabefelder für die GUI klar definiert. Am Ende kommt es darauf an, wie die Eingabedaten der Benutzeroberfläche ausgewertet werden. Das Programm *openssl* fungiert als eine Art Back-End, wo das Kommando als textuelle Form übergeben wird. Daher ist es wichtig, dass die entsprechende Logik für den Zusammenbau eines Kommandos funktioniert und in ein gültiges Format überträgt.

Die Entwicklung des Front-Ends der Web-Applikation hat gezeigt, dass die Realisierung einer Benutzeroberfläche für OpenSSL mit den Web-Technologien HTML, CSS und JS problemlos möglich ist. Damit werden die notwendigen Werkzeuge zur Verfügung gestellt, um eine funktionale Anwendung unter Berücksichtigung der Anforderungen (siehe Kap. 3.2) zu gestalten. Als einzige Voraussetzung gilt eine Umgebung mit dem ausführbaren Programm *openssl*.

Die gestellten funktionalen Anforderungen (Kap. 3.2.1) an die Web-Applikation wurden alle umgesetzt und zeigen sich als funktionsfähig. Der Benutzer ist somit in der Lage auf zwei verschiedene Arten mit OpenSSL zu interagieren. Hinsichtlich der Zuverlässigkeit, unterscheidet sich jedoch die grafische Implementierung einzelner Kommandos zu der dargestellten Kommandozeile. Da die Eingabefelder der GUI im Voraus bekannt sind, können diese vor der Ausführung validiert werden. Damit besteht keine Gefahr, dass der Nutzer ungültige Eingaben tätigt. Hingegen ist die Verwendung der Kommandozeile uneingeschränkt, das heißt, die dort eingegebenen Kommandos werden nicht überprüft und direkt an das Wasm-Modul weitergeleitet. Als Folge dessen besteht die Möglichkeit, dass der Nutzer nicht unterstützte Kommandos aufruft.

Verwendung des OpenSSL-Toolkits im Browser

Trotz der erfolgreichen Kompilierung ergeben sich einige Einschränkungen hinsichtlich der Verwendung im Web-Browser. Die gebotenen Funktionalitäten des OpenSSL-Toolkits sind umfangreich, sodass sich die Web-Applikation primär auf die Grundfunktionen wie Verschlüsselung, RSA-Schlüsselgenerierung und Hashes konzentriert. Da jedoch die Möglichkeit geboten wird in der Anwendung eine Kommandozeile zu nutzen, lassen sich somit zahlreiche OpenSSL-Kommandos testen.

Nimmt man *OpenSSL Cookbook* [21] als Referenz für die Verwendung der Kommandozeile, lässt sich feststellen, dass vor allem TLS-Funktionalitäten wie Client- und Server-Tests in der Web-Applikation nicht unterstützt werden. Das `s_client`-Kommando führt zu einer Fehlermeldung seitens OpenSSL. Dieser Fehler lässt sich nur schwer identifizieren ohne den Source-Code von OpenSSL zu analysieren.

Auch zwischen den verwendeten OpenSSL-Version 1.1.1k und der 3.0.0-Alpha15 machen sich Unterschiede in der Benutzung bemerkbar. Ein Beispiel dafür ist das neue Provider-Konzept in der Version 3.0. Dieses verlangt für die Nutzung von Legacy-Algorithmen einen separaten Provider, der durch eine Konfigurationsdatei im Installationsverzeichnis von OpenSSL konfiguriert wird. Eine solche Installationsumgebung gibt es bei der Ausführung des WebAssembly-Moduls nicht. Das hat zur Folge, dass der Funktionsumfang geringer ausfällt als im Vergleich zu Version 1.1.1k, da in dieser Version alle Implementierungen von kryptografischen Algorithmen in einer Bibliothek zusammengefasst sind (siehe Abb. 1). Als Beispiel für einen Legacy-Algorithmus in OpenSSL gilt die RC4-Stromchiffre. Die verwendeten Kommandos in Kapitel 2.1 zeigen sich hingegen in

beiden Versionen als funktionsfähig und bieten damit die Grundlage für das Front-End der Web-Applikation.

Ein weiterer Unterschied, der sich zwischen dem Stable- und Alpha-Release von OpenSSL in der Web-Applikation bemerkbar macht, ist die Ausführungszeit für die Berechnung eines privaten RSA-Schlüssels. Abbildung 13 zeigt eine Messreihe der Berechnungszeiten von einem Schlüssel mit der Länge 4096 bit. Die Messung wurde mit den Developer-Tools im Web-Browser Chromium durchgeführt und als Messwert galt die Berechnungszeit der Runtime-Methode `callMain()`.

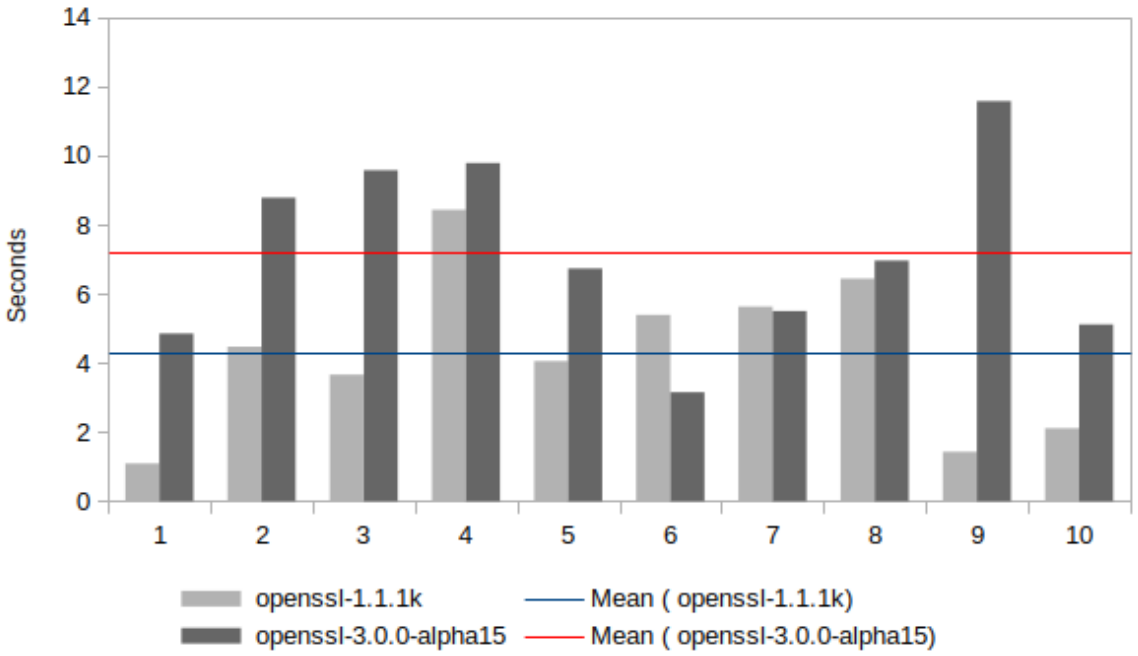

Abbildung 13: Berechnungszeiten eines RSA-Schlüssels der Länge 4096 bit

Das Diagramm in Abbildung 13 macht deutlich, dass die durchschnittliche Berechnungszeit mit der Version 3.0-Alpha15 rund drei Sekunden mehr beträgt. Das hängt mit den Änderungen in Version 3.0 zusammen, da für die Primzahlerzeugung fortan mindestens 64 Runden des Miller-Rabin-Primzahltests durchlaufen werden. [5] Daraus resultiert eine erhöhte Schlüsselgenerierungszeit. Für die Web-Applikation ist dies aus folgendem Grund relevant: Jede Ausführung durch das Wasm-Modul blockiert den Main-Thread des Browsers. Das heißt, der Benutzer ist während der Berechnung eines RSA-Schlüssels nicht in der Lage, mit der Web-Applikation zu interagieren. Das Blockieren der GUI ist störend und sollte möglichst vermieden werden. Aufgrund dessen ist Version 1.1.1k die bessere Wahl für die Web-Applikation, um die Blockierung des Main-Threads bei einer RSA-Schlüsselgenerierung so gering wie möglich zu halten.

6. Schluss

6.1. Zusammenfassung

Im Rahmen dieser Arbeit wurde eine benutzerfreundliche Web-Applikation entwickelt zur Client-seitigen Verwendung der Open-Source-Software OpenSSL. Die Grundlage dafür bildeten Web-Technologien wie WebAssembly, HTML, CSS und JS.

Zunächst wurden die zwei Szenarien (Kap. 3.1) einer möglichen Umsetzung im Web analysiert. Daraus ergab sich eine benutzerfreundliche Variante in Form eines Graphical User Interface (GUI), um die Interaktion mit OpenSSL interaktiv zu gestalten. Des Weiteren wurde eine Variante als Kommandozeile vorgestellt, die zur Verwendung von OpenSSL im klassischen Sinne dient. Für die Web-Applikation waren beide Varianten vorgesehen und sollen dem Nutzer eine Auswahl bieten mit dem OpenSSL-Toolkit zu interagieren.

Die Implementierung der Web-Applikation erfolgte auf Grundlage der in der technischen Analyse (Kap. 3.4) evaluierten Kerntechnologien Emscripten und React. Mit Hilfe der Emscripten-Compiler-Toolchain wurde das OpenSSL-Projekt erfolgreich zu einem WebAssembly-Modul kompiliert und legte damit den Grundstein für die weitere Entwicklung der Anwendung. Um die Funktionalitäten des Programms *openssl* über ein Front-End bereitzustellen, wurde das JS-Framework React gewählt. Die damit verbundenen Vorteile einer Single Page Application (SPA) tragen dazu bei, dass sich die Web-Applikation wie eine native Anwendung im Browser verhält.

Diese Arbeit hat gezeigt, dass sich das Kompilierungsziel WebAssembly mit Hilfe von Emscripten dafür eignet, um die Programmbibliotheken von OpenSSL für eine Anwendung im Web Client-seitig bereitzustellen. Erst durch Emscripten werden alle notwendigen Werkzeuge geboten, sodass ein C-Projekt wie OpenSSL erfolgreich zu WebAssembly kompiliert werden kann. Eine genauso wichtige Rolle spielt Emscripten hinsichtlich der Ausführung im Web-Browser. Die erforderlichen Schnittstellen für Systemaufrufe werden von Emscripten mittels JavaScript an die Browserumgebung angepasst und bietet damit eine funktionsfähige Laufzeitumgebung für OpenSSL als WebAssembly-Format.

Darauf aufbauend wurde eine Möglichkeit dargestellt, wie sich das Kommandozeilenprogramm aus OpenSSL auf eine benutzerfreundliche Art im Web-Browser verwenden lässt. Die OpenSSL-Befehle erwiesen sich als gut strukturiert und damit als optimale Voraussetzung für die Interpretation durch eine grafische Oberfläche. Web-Technologien wie HTML, JS und CSS verhalfen dazu, die Anwendung in einer Browserumgebung darzustellen und den Befehl an das WebAssembly-Modul zu übergeben. Letztlich ist die Gestaltung einer Benutzeroberfläche nur davon abhängig, ob OpenSSL in der jeweiligen Host-Umgebung ausführbar ist. Für die Web-Applikation ist diese Voraussetzung gege-

ben, wodurch die kryptografischen Funktionen aus OpenSSL plattformunabhängig für die breite Nutzung im Web verfügbar werden.

6.2. Ausblick

Die Web-Applikation sollte sowohl direkt über die GUI funktionieren als auch den didaktischen Effekt erzielen, den Aufbau der Befehle indirekt zu erlernen. Zudem bietet die Anwendung die Möglichkeit, sich an der Kommandozeile auszuprobieren, ohne das Hindernis einer vorherigen Installation (sofern OpenSSL nicht vorinstalliert ist). Ein Einsatzort für solch eine Anwendung ist das CTO. Der Schwerpunkt von CTO liegt auf einer interaktiven Benutzung von Verschlüsselungsmethoden und Analyseverfahren, die zusätzlich mit verständlichen Erläuterungen zur Funktionsweise der Verfahren einen Einblick in die Kryptologie gewähren. Damit lädt die Webseite dazu ein, mit den dort vorgestellten Verfahren zu experimentieren. Durch das Plugin-System von CTO ist die Webseite zudem leicht erweiterbar und bietet die optimalen Bedingungen für eine Integration der Web-Applikation.

Die aktuelle Version der Web-Applikation ist für einen produktiven Einsatz im CTO allerdings noch nicht bereit. Dafür fehlt zum einen die deutsche Übersetzung der Benutzeroberfläche, zum anderen erfordert die Integration noch kleinere Designanpassungen, damit die Anwendung mit dem verwendeten Design in CrypTool-Online übereinstimmt. Es ist geplant, diese beiden Punkte im Anschluss an diese Arbeit umzusetzen, um die Anwendung letztendlich im CTO zu veröffentlichen.

Eine weitere Verbesserungsmöglichkeit für die Web-Applikation besteht im Zusammenhang mit der folgenden Problematik aus Kapitel 5. Die Blockierung des Main-Threads bei der RSA-Schlüsselgenerierung verhindert die Interaktion mit der GUI. Hier lohnt es sich das Wasm-Modul als Web Worker zu integrieren, um die Berechnungen nebenläufig durchzuführen. Somit wird der Main-Thread für UI-Interaktionen freigehalten.

Darüber hinaus muss erwähnt werden, dass die aktuelle Implementierung der Web-Applikation längst nicht den gesamten Umfang des OpenSSL-Toolkits widerspiegelt. Bspw. implementiert die grafische Umsetzung des `dgst`-Befehl (siehe Abb. 16) lediglich eine von möglichen 25 Optionen des Befehls. Denkbar wäre es, dass man die grafischen Implementierungen zu den Kommandos weiter ausbaut. Auch das Hinzufügen weiterer Kommandos als GUI, wäre eine Möglichkeit zur Erweiterung der Web-Applikation. Beispielsweise würde sich das `x.509`-Kommando zur Verwaltung von digitalen Zertifikaten anbieten.

Das soll als Motivation für eine zukünftige Weiterentwicklung der in dieser Arbeit vorgestellten Web-Applikation dienen.

A. Komponenten der Web-Applikation

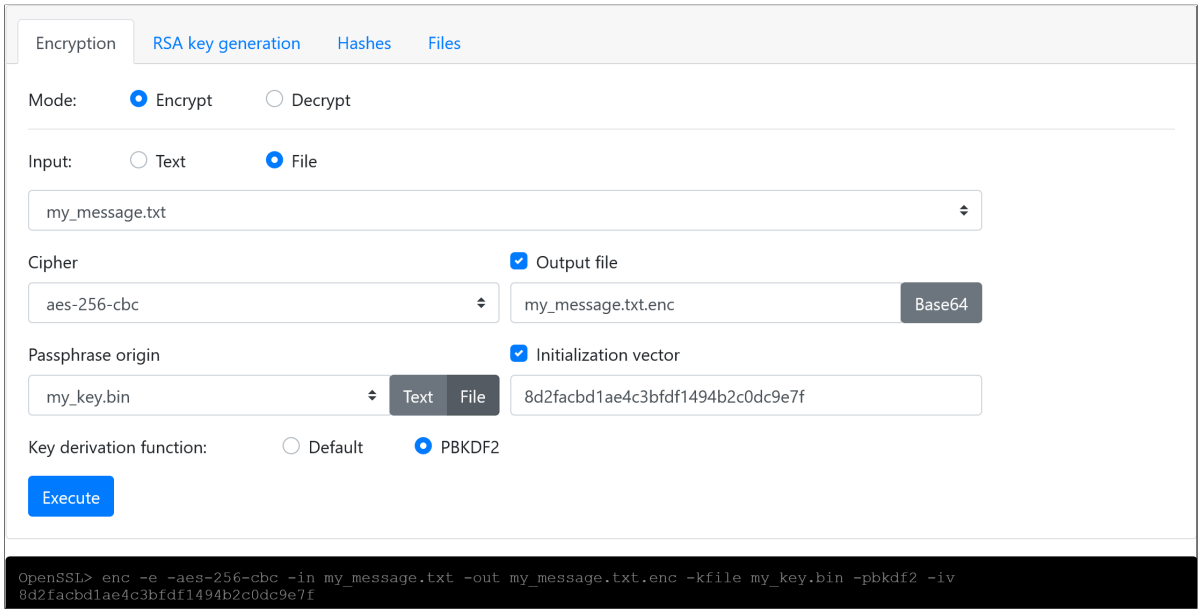

Abbildung 14: TabEncryption-Komponente

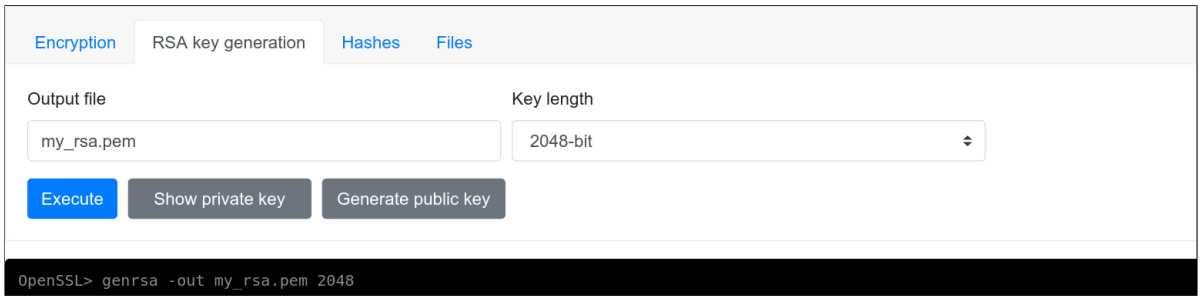

Abbildung 15: TabGenrsa-Komponente

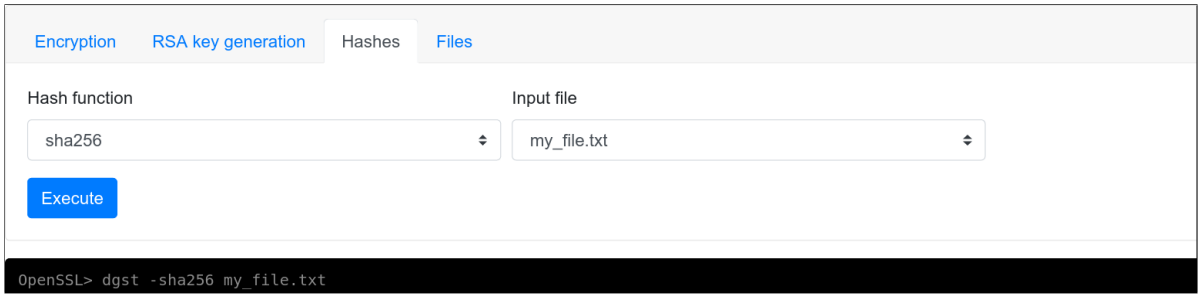

Abbildung 16: TabDigest-Komponente

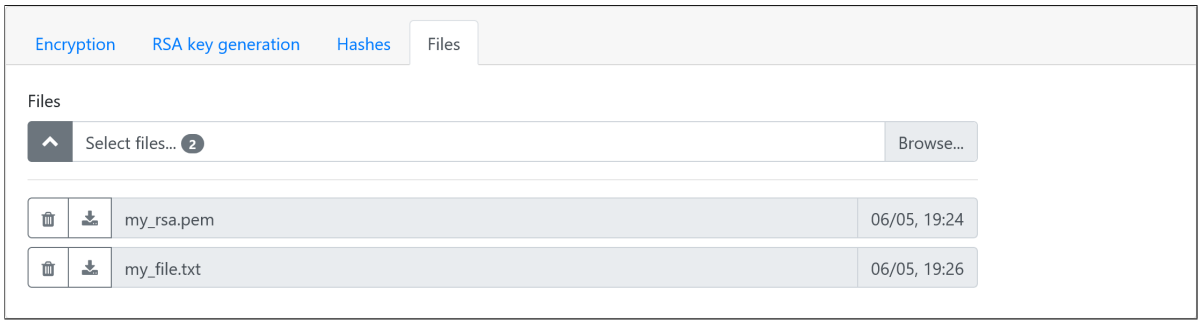

Abbildung 17: TabFiles-Komponente

Abbildung 18: CommandLine-Komponente

Literaturverzeichnis

[1] *Adobe Flash*. URL: de.wikipedia.org/wiki/Adobe_Flash#Sicherheitsl%C3%BCcken (besucht am 05. 04. 2021).

[2] Michael Braun. „Nicht-funktionale Anforderungen“. In: *Juristisches IT-Projektmanagement Lehrstuhl für Programmierung und Softwaretechnik Ludwig-Maximilians-Universität München* (2016).

[3] Peter Bühler, Patrick Schlaich und Dominik Sinner. *HTML5 und CSS3 Elektronische Ressource: Semantik - Design - Responsive Layouts*. Bibliothek der Mediengestaltung. Berlin, Heidelberg: Springer Berlin Heidelberg, Imprint: Springer Vieweg, 2017. ISBN: 978-3-662-53916-3. DOI: 10.1007/978-3-662-53916-3.

[4] *Building Projects*. URL: https://emscripten.org/docs/compiling/Building-Projects.html (besucht am 26. 04. 2021).

[5] *Changelog*. URL: https://www.openssl.org/news/changelog.html (besucht am 06. 05. 2021).

[6] *Compilation and Installation*. URL: https://wiki.openssl.org/index.php/Compilation_and_Installation (besucht am 25. 04. 2021).

[7] *Emscripten Runtime Environment*. URL: https://emscripten.org/docs/porting/emscripten-runtime-environment.html (besucht am 24. 04. 2021).

[8] Bernhard Esslinger, Hrsg. *Das CrypTool-Buch: Kryptographie lernen und anwenden mit CrypTool und SageMath*. 12. Aufl. CrypTool-Projekt, 2020.

[9] *File System Overview*. URL: https://emscripten.org/docs/porting/files/file_systems_overview.html#file-system-overview (besucht am 17. 04. 2021).

[10] Liang Gong, Michael Pradel und Koushik Sen. „JITProf: Pinpointing JIT-Unfriendly JavaScript Code“. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, S. 357–368. DOI: 10.1145/2786805.2786831.

[11] Andreas Haas u. a. „Bringing the Web up to Speed with WebAssembly“. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2017, S. 185–200. DOI: 10.1145/3062341.3062363.

[12] David Herrera, Hangfen Chen, Erick Lavoie und Laurie Hendren. „WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices“. In: *University of McGill, Montreal: QC, Technical report SABLE-TR-2018-2* (2018).

[13] *Hooks at a Glance*. URL: https://reactjs.org/docs/hooks-overview.html (besucht am 29. 04. 2021).

[14] *Introducing JSX*. URL: https://reactjs.org/docs/introducing-jsx.html (besucht am 30. 04. 2021).

[15] *OpenSSL*. URL: https://www.similartech.com/technologies/openssl (besucht am 06. 05. 2021).

[16] *OpenSSL 3.0*. URL: https://wiki.openssl.org/index.php/OpenSSL_3.0 (besucht am 22. 03. 2021).

[17] *OpenSSL Blog*. URL: https://www.openssl.org/blog/blog/2018/11/28/version/ (besucht am 30. 03. 2021).

[18] OpenSSL Software Foundation. *OpenSSL Strategic Architecture*. URL: https://www.openssl.org/docs/OpenSSLStrategicArchitecture.html (besucht am 22. 03. 2021).

[19] *Portability*. URL: https://webassembly.org/docs/portability/ (besucht am 27. 04. 2021).

[20] *Portability Guidelines*. URL: https://emscripten.org/docs/porting/guidelines/portability_guidelines.html (besucht am 05. 04. 2021).

[21] Ivan Ristić. *OpenSSL Cookbook*. London, United Kingdom: Feisty Duck Limited, 2021.

[22] Matthias Rohr. „Einleitung". In: *Sicherheit von Webanwendungen in der Praxis: Wie sich Unternehmen schützen können – Hintergründe, Maßnahmen, Prüfverfahren und Prozesse*. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 1–43. ISBN: 978-3-658-20145-6. DOI: 10.1007/978-3-658-20145-6_1.

[23] *Standardizing WASI: A system interface to run WebAssembly outside the web*. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ (besucht am 23. 04. 2021).

[24] *Thinking in React*. URL: https://reactjs.org/docs/thinking-in-react.html (besucht am 06. 04. 2021).

[25] *WASI: WebAssembly System Interface*. URL: https://github.com/WebAssembly/WASI/blob/main/docs/WASI-overview.md (besucht am 24. 04. 2021).

[26] *Welcome to the OpenSSL Project*. URL: https://github.com/openssl/openssl (besucht am 28. 03. 2021).

[27] *What is a Polyfill?* URL: https://remysharp.com/2010/10/08/what-is-a-polyfill (besucht am 05. 05. 2021).

[28] B. Yee u. a. „Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *2009 30th IEEE Symposium on Security and Privacy*. 2009, S. 79–93. DOI: 10.1109/SP.2009.25.

[29] Alon Zakai. „Emscripten: An LLVM-to-JavaScript Compiler". In: New York, NY, USA: Association for Computing Machinery, 2011, S. 301–312. DOI: 10.1145/2048147.2048224.

[30] Alon Zakai. *The History of WebAssembly*. Youtube. 3. Dez. 2020. URL: https://www.youtube.com/watch?v=XuZt1OCCQTg (besucht am 03. 04. 2021).

Abkürzungsverzeichnis

AES	Advanced Encryption Standard
API	Application Programming Interface
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CLI	Command Line Interface
CMVP	Cryptographic Module Validation Program
CSS	Cascading Style Sheets
CTR	Counter Mode
CTO	CrypTool-Online
DOM	Document Object Model
ECB	Electronic Code Book
FIPS	Federal Information Processing Standard
GnuPG	GNU Privacy Guard
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IV	Initialisierungsvektor
JIT	Just-in-time
JS	JavaScript
JSX	JavaScript XML
KDF	Key Derivation Function
LLVM	Low Level Virtual Machine
LLVMIR	LLVM Intermediate Representation
LTS	Long Term Support
MD	Message Digest
MVP	Minimum Viable Product
NaCl	Native Client
NPAPI	Netscape Plugin API
OFB	Output Feedback
PBKDF2	Password-Based Key Derivation Function 2
PPAPI	Pepper Plugin API
POSIX	Portable Operating System Interface
SDK	Software Development Kit
SDL	Simple DirectMedia Layer
SFI	Software-based Fault Isolation
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
SPA	Single Page Application
TLS	Transport Layer Security
UI	User Interface

VM	Virtuelle Maschine
W3C	World Wide Web Consortium
WASI	WebAssembly System Interface
Wasm	WebAssembly

Abbildungsverzeichnis

Listingverzeichnis

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.

10.05.2021 (Datum)


(Unterschrift)

Inhalt der CD

- Bachelorarbeit als PDF
- Quellcode der Web-Applikation