UNIVERSITÄT SIEGEN

WIRTSCHAFTSWISSENSCHAFTEN
WIRTSCHAFTSINFORMATIK | WIRTSCHAFTSRECHT

Masterarbeit
zur Erlangung des Grades Master of Science Wirtschaftsinformatik

Analyse des neuen Webstandards WebAssembly

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

vorgelegt von Thorben Groos
Matrikelnummer 1208382
Abgabedatum 21.03.2021

Zusammenfassung

Mit der Entwicklung von WebAssembly steht seit 2019 ein neuer Webstandard im Browser zur Verfügung. Er verspricht leistungsstark, sicher und portabel zu sein. Basierend auf den Versprechungen sollte sich WebAssembly gut für den produktiven Einsatz auf einer Webseite eignen. Diese Annahme wird in dieser Arbeit überprüft. Dazu wird WebAssembly in den Bereichen Geschwindigkeit, Sicherheit, Oberflächengestaltung und Implementierung analysiert und bewertet. Basierend auf den Erkenntnissen der Analysen wird eine Empfehlung gegeben, ob WebAssembly auf einer produktiven Webseite eingesetzt werden sollte. Die Analysen ergaben, dass WebAssembly deutlich schneller ist als JavaScript, dass es ähnlich sicher ist wie JavaScript, dass es zur indirekten Gestaltung von Oberflächen genutzt werden kann und dass es Hürden bei der Implementierbarkeit gibt. Abschließend kommt die Arbeit zu dem Schluss, dass WebAssembly, insbesondere aufgrund seiner Leistungsfähigkeit, in produktiven Webseiten eingesetzt werden sollte.

Abstract

With the development of WebAssembly, a new web standard is available in the browser. It promises to be powerful, secure and portable. Based on these promises, WebAssembly should be well suited for productive use on a website. This assumption will be tested in this paper. For this purpose, WebAssembly is analyzed and evaluated in the areas of speed, security, interface design and implementation. Based on the findings of the analyses, a recommendation is made as to whether WebAssembly should be used on a productive website. The analyses showed that WebAssembly is significantly faster than JavaScript, that it is similarly secure as JavaScript, that it can be used for the indirect design of interfaces and that there are obstacles in its implementability. Finally, the study concludes that WebAssembly should be used in productive websites, especially because of its performance.

Inhaltsverzeichnis

Zusammenfassung 2

Abstract 2

Inhaltsverzeichnis 3

1 Einleitung 5
1.1 Motivation . . . 5
1.2 Ziele der Arbeit . . . 6
1.3 Aufbau der Arbeit . . . 6
1.4 Verwendete Software . . . 7

2 Grundlagen 9
2.1 Übersicht über WebAssembly . . . 9
2.2 Historie von WebAssembly . . . 10
2.3 Unterschiede zwischen WebAssembly und seinen Vorgängern . . . 11
2.4 WebAssembly-Toolchains . . . 14
2.4.1 Clang-LLVM-Toolchain . . . 16
2.4.2 Emscripten-Toolchain . . . 17

3 Evaluierung der Geschwindigkeit von WebAssembly 19
3.1 Theoretische Untersuchung der Geschwindigkeit . . . 20
3.1.1 Download-Zeiten . . . 22
3.1.2 Parsing- / Dekodierungszeiten . . . 23
3.1.3 Kompilierungs- und Optimierungszeiten . . . 26
3.1.4 Re-Optimierungszeiten . . . 29
3.1.5 Laufzeiten . . . 30
3.1.6 Garbage Collection-Zeiten . . . 31
3.2 Praktische Untersuchung der Geschwindigkeit . . . 32
3.2.1 Versuchsaufbau . . . 33
3.2.2 Testumgebung . . . 34
3.2.3 Ladegeschwindigkeiten . . . 34
3.2.4 Ausführungsgeschwindigkeiten . . . 41
3.3 Fazit . . . 43

4 Evaluierung der Sicherheit von WebAssembly 45
4.1 Allgemeine Informationen zur Sicherheit . . . 45
4.2 Speichersicherheit . . . 46
4.3 Control Flow Integrity . . . 49
4.4 In-Browser Crypto Mining . . . 51
4.5 Obfuskation von Schadcode . . . 53
4.6 Fazit . . . 53

5 Möglichkeiten zur Oberflächengestaltung mit WebAssembly 55
- 5.1 Allgemeine Informationen zur Oberflächengestaltung 55
- 5.2 Indirekte Manipulation des DOM durch WebAssembly 56
- 5.3 Fazit 57

6 Erkenntnisse aus der Implementierung der Beispiele 58
- 6.1 Wahl der Toolchain 58
- 6.2 Instantiierung von WebAssembly 60
- 6.3 Hindernisse bei der Portierung vorhandener Codebasen zu WebAssembly 62
- 6.4 Exception Handling 63
- 6.5 Speicherreservierung und -nutzung 65
- 6.6 Optimierung von JS-Code als Alternative zu WebAssembly 67
- 6.7 Geschwindigkeit von WebAssembly in verschiedenen Browsern 68
- 6.8 Fazit 70

7 Related Work 73

8 Zusammenfassung und Ausblick 74

Literaturverzeichnis 76

Abkürzungsverzeichnis 84

Abbildungsverzeichnis 85

Code-Listings 86

Anhang 87

Eidesstattliche Erklärung 88

1 Einleitung

Die Entwicklung des World Wide Web geht mit großen Schritten voran. Damit einhergehend steigt die Nachfrage nach leistungsstarken Websites, um rechenintensive Aufgaben wie 3D-Rendering, Bild- und Videobearbeitung, Softwareentwicklung (inklusive Kompilieren und Debuggen) oder Verschlüsselungen im Browser auszuführen [Chm+20; W3C20e]. Diese Anwendungsfälle erfordern die Möglichkeit, Code sicher, schnell sowie plattform- und hardwareunabhängig ausführen zu können [Haa+17].

Bisherige Webtechnologien, wie JavaScript (JS), ActiveX, Native Client (NaCl) oder asm.js konnten die Anforderungen nicht gleichzeitig erfüllen. Die Möglichkeit, rechenintensive Aufgaben im Browser auszuführen, war daher begrenzt [Haa+17].

1.1 Motivation

Seit Dezember 2019 gibt es einen neuen Standard in der Welt der Webtechnologien. Er verspricht, die Anforderungen an Sicherheit, Schnelligkeit und Portabilität zu erfüllen: WebAssembly (Wasm) [W3C19]. WebAssembly ist ein Low-Level-Bytecode, der in der JavaScript Virtual Machine (JSVM) ausgeführt wird. Er soll annähernd native Performance im Browser erreichen. Durch die Nutzung der JSVM verspricht er so sicher wie JS zu sein [Haa+17]. Weiterhin soll er plattform- und hardwareunabhängig sein. Wasm soll so die Verlagerung rechenintensiver Aufgaben in den Browser ermöglichen. Er ist bereits als Minimum Viable Product (MVP) in den vier großen Browser Firefox, Chrome, Edge und Safari implementiert [W3C20g]. Das macht ihn für den Einsatz in der Praxis interessant.

Erwartungsgemäß ist das Web gefüllt mit positiven Artikeln über Möglichkeiten, Leistungsfähigkeit, Portabilität und Sicherheit von WebAssembly. Sie sind häufig mit minimalisierten Beispielimplementierungen verbunden, welche die beschriebenen Sachverhalte verdeutlichen und die einfache Implementierbarkeit hervorheben sollen. Viele minimale Beispiele haben jedoch nur annähernd etwas mit realen Anwendungsfällen gemein. Folglich können nur bedingt Rückschlüsse auf praxisnahe Anwendungen gezogen werden.

Auffällig ist die Abwesenheit negativer Artikel, die Nachteile und Grenzen von Wasm zeigen. Es entsteht der Eindruck, dass es keine Nachteile gibt. Diese Vermutung gilt es aber anzuzweifeln. Es ist daher von Interesse, die Möglichkeiten und Grenzen von WebAssembly aus einer theoretischen und praktischen Sicht zu analysieren, um neben den Vorteilen auch die Nachteile von WebAssembly zu erkennen.

1.2 Ziele der Arbeit

Ziel der Masterarbeit ist es, eine Bewertung des Webstandards WebAssembly für den praktischen Einsatz zu geben. Dazu sollen zu Beginn die Grundlagen von WebAssembly dargestellt werden. Die Grundlagen zu kennen, erhöht die Nachvollziehbarkeit der folgenden Kapitel. Daraufhin soll WebAssembly in den Bereichen Geschwindigkeit, Sicherheit und Oberflächengestaltung evaluiert werden. Die Evaluierungen sollen zeigen, was bei einem Einsatz von WebAssembly aus Sicht des jeweiligen Bereiches beachtet werden muss, und, ob sich WebAssembly aus Sicht des Bereiches für einen praktischen Einsatz eignet. Die bei der Evaluierung gesammelten Erkenntnisse und Erfahrungen sollen aufbereitet werden. Darauf aufbauend sollen Empfehlungen für die Programmierung mit WebAssembly gegeben werden. Basierend auf den Ergebnissen der drei Analysen und der Sammlung der Erkenntnisse und Erfahrungen soll eine bereichsübergreifende Bewertung erstellt werden, ob sich WebAssembly für den praktischen Einsatz eignet oder nicht.

Zusammenfassend sind folgende Ziele zu erreichen:

Ziel 1: Darstellung der Grundlagen von WebAssembly

Ziel 2: Evaluierung der Geschwindigkeit von WebAssembly

Ziel 3: Evaluierung der Sicherheit von WebAssembly

Ziel 4: Evaluierung der Möglichkeiten zur Oberflächengestaltung mit WebAssembly

Ziel 5: Entwicklung von Empfehlungen für die Programmierung mit WebAssembly

Ziel 6: Bewertung von WebAssembly für den praktischen Einsatz

1.3 Aufbau der Arbeit

Kapitel 2 gibt eine Einführung in die Grundlagen von WebAssembly. Es werden allgemeine Fakten über Wasm, seine Historie und die Unterschiede zu seinen Vorgängern genannt. Zusätzlich werden zwei bekannte Toolchains zur Kompilierung von Code zu WebAssembly vorgestellt.

In Kapitel 3 wird die Geschwindigkeit von WebAssembly evaluiert. Es wird ausführlich darauf eingegangen, wie Wasm- und JS-Code im Browser verarbeitet werden. Dabei wird gezeigt, wodurch die Geschwindigkeitsvorteile von Wasm im Vergleich zu JS entstehen. Die Geschwindigkeit wird anschließend anhand einer RC4-Verschlüsselung und eines

Unity-Benchmark-Projekts überprüft. Abschließend wird ein Fazit gezogen, ob WebAssembly aus Sicht der Geschwindigkeit eingesetzt werden sollte.

Die Sicherheit von WebAssembly wird in Kapitel 4 analysiert und bewertet. Zuerst wird die Sicherheit von Wasm im Allgemeinen betrachtet. Daraufhin werden spezielle sicherheitskritische Bereiche, wie die Speichersicherheit, die Control Flow Integrity, das In-Browser Crypto Mining und die Obfuskation von Schadcode, beleuchtet. Abschließend wird ein Fazit gezogen, ob WebAssembly aus Sicht der Sicherheit verwendet werden kann.

Die Möglichkeiten, Website-Oberflächen mit WebAssembly zu gestalten, werden in Kapitel 5 evaluiert. Dabei wird die indirekte Manipulation von Weboberflächen beschrieben. Sie wird anschließend anhand einer in WebAssembly berechneten Animation verdeutlicht. Das Kapitel endet mit einem Fazit, das die Eignung von WebAssembly zur Gestaltung von Oberflächen im Browser beinhaltet.

In Kapitel 6 werden die während der Implementierung der verwendeten Beispiele gewonnenen Erkenntnisse gesammelt und aufgearbeitet. Es werden Probleme, Dos und Donts, sowie weitere interessante Erkenntnisse im Zusammenhang mit der Programmierung mit WebAssembly aufgeführt.

Kapitel 7 zeigt nennenswerte Arbeiten im Umfeld dieser Arbeit. Die wichtigsten Artikel und Autoren werden mit ihrer Bedeutung für diese Arbeit genannt.

Die Arbeit endet in Kapitel 8 mit einer Zusammenfassung der Ergebnisse und einer abschließenden Bewertung, ob WebAssembly für den produktiven Einsatz geeignet ist oder nicht. Anschließend findet ein Ausblick auf weiterführende Arbeiten statt.

1.4 Verwendete Software

Die Entwicklung um WebAssembly ist sehr lebhaft. Tools werden häufig aktualisiert. Sie unterstützen je nach Version unterschiedliche Funktionen. Um die Beispiele in dieser Arbeit reproduzieren zu können ist es daher wichtig zu wissen, welche Tools in welcher Version verwendet wurden. Die Tools und deren Versionen werden im Folgenden aufgelistet.

Die verwendeten Tools sind allesamt aktuelle Stable-Versionen. Beta-Versionen werden explizit nicht benutzt, um die Tools als Fehlerquellen ausschließen zu können. Die Release-Zyklen der Stable-Versionen sind sehr kurz, nur wenige Tage bis Wochen. Der Nachteil von Stable-Versionen, nicht den aktuellsten Stand abzubilden, ist daher gering. Er wird zugunsten der Stabilität der Tools hingenommen.

emsdk 2.0.9 (Release 16.11.2020): Emscripten SDK (emsdk) ist ein Paketmanager für alle Tools, die für die Emscripten-Toolchain benötigt werden. Dazu gehören das Compiler-Frontend emcc, der Compiler LLVM und das Compiler-Backend LLVM WebAssembly Backend. Zusätzlich enthält es Tools, die die Nutzung der Emscripten-Toolchain vereinfachen. Dazu gehören unter anderem Java, Python, Node.js und mingw [Ems20d].

Python 3.9.0 (Release 05.10.2020): Python ist eine universelle Programmiersprache, die verschiedene Module zur Verfügung stellt. Eines der Module ist ein einfacher HTTP-Server. Er wird genutzt, um HTML-, JS- und Wasm-Dateien lokal bereitzustellen. Ein aktueller Server ist wichtig, da er .wasm-Dateien mit dem MIME-Type *application/wasm* bereitstellen muss. Die Instantiierung von WebAssembly mit der performantesten Funktion ist nur mit einem korrekten MIME-Type möglich.

Firefox 82.0.2 (Release 28.10.2020): Firefox ist ein freier Webbrowser. Er wird benötigt, um den JS- und Wasm-Code auszuführen. Es wird Firefox genutzt, da dieser Stand Oktober 2020 die meisten und neusten Features von WebAssembly unterstützt [W3C20d].

Clang 11.0.0 (Release 12.10.2020): Clang ist ein Compiler-Frontend für den Compiler LLVM. Es kann Sprachen der C-Familie in die LLVMIR übersetzen. Die LLVMIR ist ein Zwischencode, der von LLVM genutzt wird. Dieser Schritt ist notwendig, um C-Code zu WebAssembly zu kompilieren.

LLVM 11.0.0 (Release 12.10.2020): LLVM ist ein modularer Compiler. Er erlaubt die Optimierung von LLVMIR-Code über alle Laufzeitphasen eines Programms hinweg. Dadurch ist eine ganzheitliche Optimierung möglich.

LLVM WebAssembly Backend 11.0.0 (Release 12.10.2020): Das LLVM WebAssembly Backend ist ein Compiler-Backend von LLVM. Es kompiliert Code aus der LLVMIR in WebAssembly. Das Backend ist kein eigenständiges Tool, sondern ist ein Teil von LLVM. Es wird mit LLVM ausgeliefert und besitzt daher dieselbe Versionsnummer und dasselbe Releasedatum wie LLVM.

2 Grundlagen

WebAssembly (Wasm) ist eine junge Technologie, deren Grundlagen noch weitgehend unbekannt sind. Kenntnisse darüber zu besitzen ist jedoch nötig, um die Vor- und Nachteile von Wasm in den folgenden Kapiteln vollständig nachvollziehen zu können. In diesem Kapitel werden daher die Grundlagen von WebAssembly vorgestellt.

Abschnitt 2.1 gibt einen Überblick über WebAssembly. Es zeigt den Nutzen von Wasm und ordnet es zwischen bestehenden Webtechnologien ein. Abschnitt 2.2 fokussiert die Geschichte von Wasm seit seiner erstmaligen Veröffentlichung. Im darauffolgenden Abschnitt 2.3 werden Unterschiede zwischen WebAssembly und vorhergehenden, ähnlichen Technologien aufgeführt. Dabei wird das Potenzial von Wasm im Vergleich zu diesen sichtbar. Abschließend thematisiert Abschnitt 2.4, mithilfe welcher Tools C und C++ Code zu WebAssembly kompiliert werden kann. Dabei wird darauf eingegangen, warum nicht direkt von C zu WebAssembly kompiliert wird, sondern ein Zwischenschritt gemacht wird.

Hinweis zur verwendeten Literatur: Wasm ist eine neue Technologie, zu der es nur wenig Literatur in Form von wissenschaftlichen Veröffentlichungen gibt. Der Hauptteil der Literatur stammt daher aus Artikeln im Web. Da die Qualität der Artikel dort schwankt, stammt wurde nur Literatur aus renommierten Quellen verwendet. Renommierte Quellen sind beispielsweise das Mozilla Developer Network (MDN), das World Wide Web Consortium (W3C) und Personen, die an der Entwicklung von Wasm beteiligt sind.

2.1 Übersicht über WebAssembly

WebAssembly ist ein Bytecode, der für die Ausführung in Browsern optimiert ist. Er wurde so designt, dass Code performant, sicher und plattformunabhängig ausgeführt werden kann. Diese Merkmale ermöglichen das Ausführen rechenintensiver Aufgaben im Browser. Das erlaubt eine Verlagerung diverser Anwendungen dorthin. Mögliche Anwendungen sind CAD- und Peer-to-Peer-Anwendungen, sowie Spiele [W3C20e]. Im Oktober 2020 unterstützen 93,2% aller Desktop-PC-Browser und 93,87% aller mobilen Browser WebAssembly [can20a].

Wasm ist keine Sprache, die von Entwicklern geschrieben wird, sondern ein Kompilierungsziel für bestehende Sprachen. Ursprünglich sollte C- und C++-Code nach WebAssembly kompiliert werden können [W3C17b]. Mittlerweile können auch andere Sprachen, wie C# oder Java, nach Wasm kompiliert werden. Dafür werden entsprechende Frameworks wie Blazor oder JWebAssembly benötigt.

Abbildung 1: WebAssembly-Logo

Wasm ist Bytecode für eine virtuelle Stack Machine. Damit die Stack Machine auf einer Vielzahl von Betriebssystemen und Prozessorarchitekturen effizient läuft, wurden minimale Anforderungen an die Laufzeitumgebung definiert. Herkömmliche, aktuelle Systeme erfüllen diese Anforderungen, sodass WebAssembly dort ausgeführt werden kann. Auf alten oder ungewöhnlichen Systemen kann es jedoch zu Problemen kommen. WebAssembly kann dort möglicherweise nicht ausgeführt werden [W3C17a].

Der Source Code von WebAssembly steht offen im WebAssembly GitHub Repository zur Verfügung [Git20].

2.2 Historie von WebAssembly

Die Entwicklung von WebAssembly wurde erstmals im Juni 2015 angekündigt [Wag15]. Ein Jahr später erschienen die ersten Demonstrationen anhand von Angry Bots, einem Tutorial-Projekt von Unity, in Mozilla Firefox [Wag16], Google Chrome [Tho16] und Microsoft Edge [Zhu16]. Daraufhin wurde Wasm in einer Browser-Preview-Phase in den Browsern mitgeliefert. Es konnte von Entwicklern und Anwendern aktiviert und getestet werden. So konnten erste Praxiserfahrungen gesammelt werden. Sie flossen in die weitere Entwicklung von Wasm ein. Die Browser-Preview-Phase wurde 2017 erfolgreich abgeschlossen und das Minimum Viable Product (MVP) von Wasm als vollständig verkündet [Wag+17]. Ab diesem Zeitpunkt ist WebAssembly rückwärtskompatibel und kann von Browserherstellern in ihren Browsern standardmäßig aktiviert werden. Seit Ende 2017 ist Wasm in allen vier großen Browsern aktiviert [McC17]. 2018 veröffentlichte die W3C WebAssembly Working Group drei Entwürfe, in denen der WebAssembly Core [W3C18b], die JavaScript API [W3C18c] und die Web API [W3C18d] spezifiziert wurden. Nach mehreren Überarbeitungen der Core Spezifikationen wurde WebAssem-

bly im Dezember 2019 ein offizieller Webstandard des World Wide Web Consortiums [W3C19].

Die Entwicklung von WebAssembly fand von 2015 bis 2017 in einer W3C Community Group statt. Nach dem Ende der Browser-Preview-Phase wurde die Entwicklung an eine W3C Working Group übergeben. Sie treibt die Standardisierung von WebAssembly seitdem voran. In beiden Gruppen wirken Mitglieder von namhaften Unternehmen, wie Mozilla, Google, Microsoft und Apple, mit [W3C20c; W3C20f].

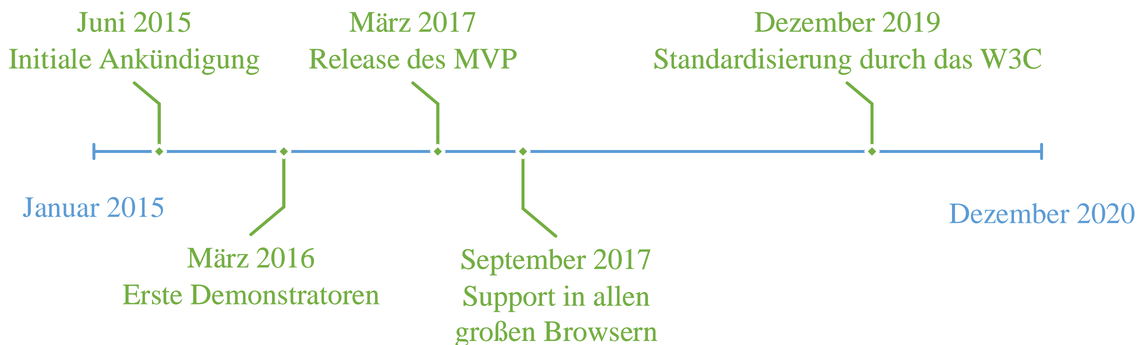


Abbildung 2: Meilensteine der WebAssembly-Entwicklung

Die Entwicklung von Wasm ist mit der Standardisierung 2019 nicht abgeschlossen worden. Sie wird anhand einer Roadmap, weitergeführt [W3C20d]. Die Roadmap listet Features auf, die WebAssembly unterstützen soll. Je nach Browser-Engine variiert die Anzahl an Features, die bereits implementiert sind. Zum Stand Oktober 2020 bieten die Engines von Firefox (Gecko) [MDN20b] und Chrome (Blink) [Chr20] die meisten unterstützen Features an. Safaris Engine WebKit [App20] steht dieser Entwicklung nach. Beispiele für Features sind die bidirektionale Konvertierung von JS BigInt zu 64 bit Integer, Exception Handling, die Unterstützung von Threads oder Single Instruction Multiple Data (SIMD) [W3C20d]. An der Konzeptionierung und Entwicklung sind weiterhin Entwickler von Mozilla, Google, Microsoft und Apple beteiligt [W3C20c].

2.3 Unterschiede zwischen WebAssembly und seinen Vorgängern

WebAssembly ist Bytecode, der im Browser ausgeführt werden kann. Die Idee, Bytecode im Browser auszuführen, ist nicht neu. Es gab in der Vergangenheit bereits mehrere Ansätze, das zu tun. Bekannte Beispiele sind ActiveX, Java-Applets, Flash, (Portable) Native Client ((P)NaCl) und asm.js [Haa+17]. Keiner der genannten Ansätze hat sich jedoch durchgesetzt [Goo19; MDN20a].

Im Folgenden werden zwei Ansätze, Googles (P)NaCl und Mozillas asm.js, vorgestellt. Sie gelten als direkte Vorgänger von Wasm [Haa+17]. Es werden Gemeinsamkeiten und Unterschiede zwischen ihnen und Wasm aufgezeigt. Dabei wird dargelegt, wo die bisherigen Ansätze Mängel haben und wie WebAssembly diese vermeidet.

Googles Native Client (NaCl) ist eine Technologie für die Ausführung von bereits kompiliertem C- und C++-Code in Google Chrome [Goo20b]. Der Code wird sicher und schnell ausgeführt [Goo20c]. NaCl läuft auf den Systemarchitekturen x86, ARM und MIPS. Dabei unterstützt es hoch performante architekturspezifische Instruktionen, wie Atomics oder SIMD. Der Code muss im Vorhinein für jede Architektur kompiliert werden. Er ist folglich nicht portabel. Native Client kann ausschließlich in Chrome Web Apps verwendet werden. Das beschränkt es auf den Chrome Browser. Weiterhin kann NaCl-Code nicht über JS in eine Website eingebunden werden. Das erschwert die Nutzung von NaCl in bereits vorhandenen Websites. WebAssembly kann im Gegensatz dazu über JS eingebunden werden. Ausgeführt wird NaCl-Code in einer Sandbox. So liefert er dieselbe Sicherheit wie JavaScript [Goo20b]. Mit einem vergleichbaren Mechanismus wird die sichere Ausführung von Wasm-Code erreicht. Weitere Informationen zur Sicherheit von Wasm sind in Kapitel 4 dargestellt.

Portable Native Client (PNaCl) ist der Nachfolger von NaCl. Er gleicht einige Unzulänglichkeiten von NaCl aus. PNaCl-Code wird nicht mehr direkt in die Maschinensprache kompiliert, sondern in eine Intermediate Representation (IR). Von der IR wird er dann vom Browser in den passenden Maschinencode für die Architektur des Anwenders kompiliert. Dadurch ist der PNaCl-Code, im Gegensatz zu NaCl-Code, portabel [Goo20a]. Eine weitere Verbesserung ist die Möglichkeit, PNaCl-Code über JavaScript einbinden zu können. Das vereinfacht die Distribution von PNaCl-Code, für den bisher der Chrome Web Store erforderlich war. Die Beschränkung auf den Chrome Browser existiert jedoch weiterhin [Goo20a]. Ein weiterer Nachteil von Portable Native Client (PNaCl) ist, dass kein synchroner Zugriff auf JavaScript oder Web APIs möglich ist [Haa+17].

Die Entwicklung von NaCl und PNaCl wurde Ende 2019 zugunsten von WebAssembly eingestellt. Auch der (P)NaCl Support im Chrome Browser wurde zu diesem Zeitpunkt beendet. Google begründete die Entscheidung wie folgt:

> „Given the momentum of cross-browser WebAssembly support, we plan to focus our native code efforts on WebAssembly going forward and plan to remove support for PNaCl in Q4 2019 (except for Chrome Apps). We believe that the vibrant ecosystem around WebAssembly makes it a better fit for new and existing high-performance web apps and that usage of PNaCl is sufficiently low to warrant deprecation.“ [Goo19]

Als weiterer Vorgänger von Wasm gilt asm.js. asm.js ist eine Untermenge von JavaScript. Daher kann es von allen Browsern ausgeführt werden, die JS unterstützen. asm.js wurde

von der Mozilla Foundation entwickelt und erstmalig 2013 erwähnt [Wag13]. Es ist ein Kompilierungsziel für statisch typisierte Sprachen, wie C, C++ oder Rust. Die Kompilierung erfolgt bspw. für C ähnliche Sprachen durch den Emscripten-Compiler. Nach der Kompilierung zu asm.js läuft der Code mit ca. der Hälfte der nativen Geschwindigkeit. Die Geschwindigkeitsvorteile gegenüber JavaScript stammen großteils aus der Typisierung und dem manuellen Speichermanagement von asm.js. Code kann dadurch besser vom Compiler und der JSVM optimiert werden. Typisierung und manuelles Speichermanagement sind Quellen, aus denen auch WebAssembly einen Teil seiner Performance zieht. Weitere Geschwindigkeitsvorteile bei asm.js sind möglich, wenn die JSVM des Browsers asm.js-Code erkennt und speziell behandelt [MDN20a]. Das ist bei Firefox, Edge und Chrome der Fall [can20b]. Wasm wird von allen großen JSVMs erkannt und speziell behandelt [can20a]. Dadurch ist die Performance von Wasm im Vergleich zu JS höher [Cla17f].

Neben der höheren Ausführungsgeschwindigkeit, die asm.js im Vergleich zu JS bietet, besitzt es jedoch auch zwei große Nachteile. Beide Nachteile wurden im Design von WebAssembly berücksichtigt und so vermieden.

Nachteil Nr. 1 ist die Zeit, die zum Parsen des asm.js-Codes gebraucht wird. asm.js-Code ist JS-Code und muss von der JSVM in einen abstrakten Syntaxbaum (engl. Abstract Syntax Tree (AST)) konvertiert werden. Dieser Vorgang kann auf mobilen Geräten bei großen Codemengen 20 bis 40 Sekunden dauern [W3C20b]. Wasm vermeidet diesen Schritt komplett, da der Code nicht in einen AST konvertiert werden muss, sondern bereits als Bytecode vorliegt [Cla17f].

Nachteil Nr. 2 ist ein Zielkonflikt zwischen guter Ahead-of-time (AOT)-Kompilierbarkeit und guter Performance in JSVMs, die nicht für asm.js optimiert sind. Dieser Konflikt erschwert das Hinzufügen performancesteigernder Features [W3C20b]. In WebAssembly wird dieser Zielkonflikt gelöst, indem eine der Anforderungen wegfällt. Wasm ist ein eigener Standard und muss daher nicht auf die Performance in nicht optimierten JSVMs achten. Entweder eine JSVM unterstützt Wasm und der Code läuft oder sie unterstützt Wasm nicht und der Code läuft nicht [MDN20e].

Die Entwicklung von asm.js wurde eingestellt und Mozilla rät von der Verwendung ab, zugunsten von WebAssembly [MDN20a]. Es kann jedoch als Fallback-Option für Wasm genutzt werden, sofern ein Browser Wasm nicht unterstützt [Cal18].

Code im Web muss schnell, sicher und plattformunabhängig sein [Haa+17]. Die Vorgänger von Wasm haben jeweils in einem der Bereiche Defizite. (P)NaCl läuft ausschließlich in Chrome und ist daher nicht plattformunabhängig [Goo20a]. asm.js ist zwar schneller als JS, jedoch nicht schnell genug [W3C20b]. Beide Defizite haben dazu geführt, dass sich die Technologien nicht durchgesetzt haben [Goo19; MDN20a]. Die Erfahrungswerte aus den Vorgängertechnologien sind in die Entwicklung von WebAssembly eingeflossen, sodass Wasm gute Chancen hat sich langfristig im Web zu etablieren [Haa+17].

2.4 WebAssembly-Toolchains

WebAssembly ist ein Bytecode, der performant im Browser ausgeführt werden kann. Um zu verstehen, warum Bytecode performant ausgeführt werden kann, hilft es zu wissen, wo Bytecode auf dem Weg von Hochsprachen-Code zu Maschinensprachen-Code angesiedelt ist, und mit welchen Tools man dorthin gelangt. Der Weg und die entsprechenden Tools werden im Folgenden vorgestellt.

Viele Computerprogramme werden in einer Hochsprache, wie C, Python oder Java geschrieben. Hochsprachen sind für Menschen gut verständlich, da deren Syntax an menschliche Gewohnheiten angepasst ist. Computer können diese Sprachen jedoch nicht verarbeiten, da sie nur Nullen und Einsen, die Maschinensprache, „verstehen“ [Cla17c]. Hochsprachen-Code muss zuvor mit verschiedenen Tools in Maschinensprachen-Code kompiliert werden. Verschiedene Tools bringen den Hochsprachen-Code näher an den Maschinensprachen-Code heran. Alle zur Kompilierung benötigten Tools aneinandergereiht werden als Toolchain bezeichnet.

Hochsprachen-Code wird häufig nicht direkt in Maschinensprachen-Code kompiliert. Für viele Sprachen wird ein Zwischenschritt über eine Intermediate Representation (IR) gemacht. Der Zwischenschritt vereinfacht das Hinzufügen einer neuen Hoch- oder Maschinensprache zu einem Compiler. Compiler sind dadurch keine monolithischen Strukturen, welche genau auf eine Quell- und Zielsprache ausgelegt sind. Sie sind modular aufgebaut und besitzen ein Compiler-Frontend, das Hochsprachen-Code in eine IR übersetzt, ein Compiler-Middleend, das Optimierungen am Code in der IR durchführt und ein Compiler-Backend, das die IR in eine Maschinensprache übersetzt. Abbildung 3 zeigt den dreiteiligen Aufbau. Soll eine neue Hochsprache hinzugefügt werden, ist nur ein neues Compiler-Frontend nötig. Es kompiliert den Hochsprachen-Code in die IR. Von der IR gibt es bereits ein Compiler-Middleend und -Backend, welches den Code optimiert und in eine Maschinensprache übersetzt. Ohne IR müsste für jede neue Maschinensprache ein komplett neuer Compiler geschrieben werden [Cla17b].

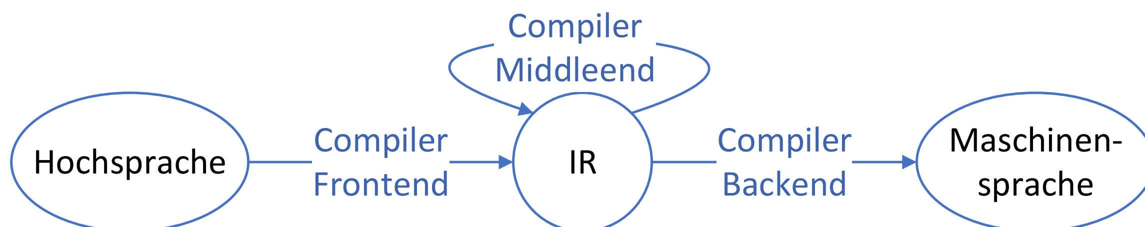


Abbildung 3: Einordnung von Compiler-Front-, -Middle- und -Backend

Abbildung 4 zeigt die Language Chain für WebAssembly. Sie enthält alle Sprachen, die auf dem Weg von Hochsprache zu Maschinensprache durchlaufen werden. WebAssembly

befindet sich zwischen IR und der Maschinensprache. Vorhandene Compiler-Frontends und -Middleends können verwendet werden, um Code in die IR zu kompilieren und dort zu optimieren. Für die Übersetzung in WebAssembly muss nur ein neues Compiler-Backend geschrieben werden. Wasm tritt aus Sicht des Compiler-Backends an die Stelle einer Maschinensprache. Die Übersetzung zwischen Wasm und der Maschinensprache der verwendeten Hardware übernimmt die JSVM [Zak16].

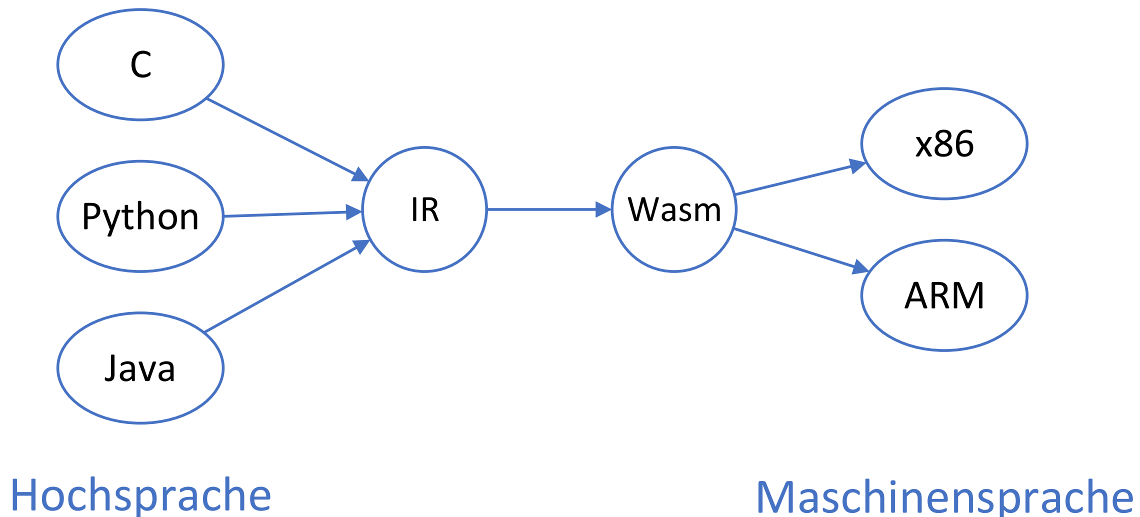


Abbildung 4: Language Chain

WebAssembly ist ein binäres Befehlsformat für eine virtuelle Maschine. Befehlssätze für virtuelle Maschinen werden als Bytecode bezeichnet. Die virtuelle Maschine für WebAssembly setzt Anforderungen an die Hardware, auf der der Code ausgeführt wird. Die Anforderungen sind das Minimum an Spezifikationen, die aktuelle, herkömmliche Hardware in PCs und Smartphones erfüllen. WebAssembly-Befehle haben eine starke Übereinstimmung mit den Maschinencode-Befehlen für die virtuelle Maschine. Die Befehle der virtuellen Maschine haben aufgrund ihrer Eigenschaft als minimale Spezifikation aktueller Hardware eine starke Übereinstimmung mit den Befehlen der schlussendlich verwendeten Hardware. Aufgrund der starken Beziehung zwischen WebAssembly-Befehlen und Maschinencode-Befehlen wird WebAssembly als Low-Level-Assembly-Like-Language bezeichnet [W3C20g]. Die Nähe zwischen WebAssembly und Maschinencode erlaubt das performante Ausführen von Code, da nur wenig Übersetzungsarbeit von der Laufzeitumgebung geleistet werden muss.

Hochsprachen-Code kann mittels unterschiedlicher Toolchains zu WebAssembly kompiliert werden. Stand Oktober 2020 gibt es für C ähnliche Sprachen zwei verbreitete Toolchains. Auf der einen Seite die Clang-LLVM-Toolchain und auf der anderen Seite die Emscripten-Toolchain. Beide Toolchains nutzen teilweise dieselben Komponenten, unterscheiden sich aber im Umfang ihrer Ausgabe. Im Folgenden wird zuerst die

Clang-LLVM-Toolchain vorgestellt, danach die bekanntere Emscripten-Toolchain. Die Reihenfolge ist so gewählt, da manche Tools aus der Clang-LLVM-Toolchain auch in der Emscripten-Toolchain genutzt werden.

2.4.1 Clang-LLVM-Toolchain

Die Clang-LLVM-Toolchain nutzt zur Kompilierung von C- und C-ähnlichem-Code zu WebAssembly das Compiler-Frontend Clang, das Compiler-Middleend LLVM und das Compiler-Backend LLVM WebAssembly Backend. Sie ist weniger bekannt als die Emscripten-Toolchain.

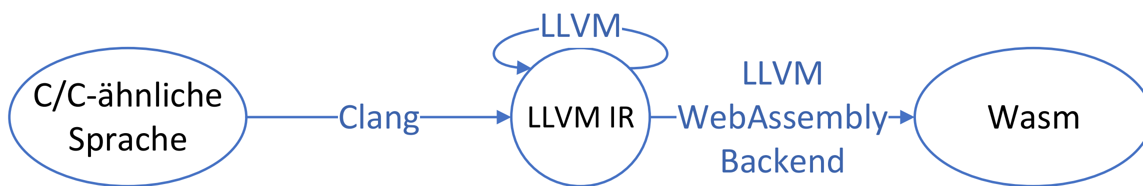


Abbildung 5: Clang-LLVM-Toolchain

Clang ist ein Compiler für C- und C-ähnliche-Sprachen (engl. C language family, kurz: Clang). Er wurde für das LLVM-Projekt als Compiler-Frontend entwickelt und kann aufgrund seiner Modularität in vielen Projekten eingesetzt werden. Er übersetzt C- und C-ähnlichen-Code in die LLVM Intermediate Representation (LLVMIR). Dort wird der Code optimiert und in eine Zielsprache kompiliert [Cla20a]. Die komplette Kompilierung von Hochsprache zu Wasm wird über Flags im Compiler-Frontend gesteuert.

Ab Release 8.0.0, veröffentlicht im im März 2019, unterstützt Clang WebAssembly standardmäßig als Zielsprache. Das erleichtert die Nutzung der Toolchain, da keine extra Flags für experimentelle Features gesetzt werden müssen [LLV19].

Ein vergleichbares Tool zu Clang ist die GNU Compiler Collection (GCC). Clang wurde als Drop-in Replacement zu GCC geschaffen. Es unterstützt die meisten Flags und Spracherweiterungen von GCC [Cla13]. Das vereinfacht das Ersetzen von GCC durch Clang, was die Umstellung auf Clang erleichtert. Dadurch wird die Kompilierung von bestehendem Code zu WebAssembly vereinfacht.

LLVM ist kein einzelnes Tool, sondern eine Sammlung an modularen und wiederverwendbaren Compilern und Toolchain-Technologien. Der Kern von LLVM ist ein quell- und zielsprachenunabhängiger Code-Optimierer. Quell- und zielsprachenunabhängig bedeutet, dass Code aus unterschiedlichen Programmiersprachen (Quellsprachen) für unterschiedliche Architekturen (Zielsprachen) optimiert werden kann. Er ist um die LLVM-

Abbildung 6: LLVM-Logo

IR herum gebaut. Die LLVMIR ist eine plattformunabhängige Assemblersprache für eine Virtuelle Maschine (VM). Die Besonderheit des Optimierers ist die Berücksichtigung sämtlicher Laufzeitphasen eines Programms bei der Optimierung. Dadurch wird Code stark optimiert [LLV20].

Der Code-Optimierer von LLVM wird auch in der Emscripten-Toolchain genutzt [Ems20c].

LLVM WebAssembly Backend ist ein Compiler-Backend. Es übersetzt die LLVMIR in WebAssembly [Ems20a].

Im Juli 2019 hat das LLVM WebAssembly Backend das zuvor in der Emscripten-Toolchain verwendete Backend fastcomp abgelöst. Es läuft schneller als fastcomp und erzeugt optimierteren Code [Zak19].

2.4.2 Emscripten-Toolchain

Die Emscripten-Toolchain (Abbildung 7) ist die bekannteste Toolchain, um C- und C++-Code zu WebAssembly zu kompilieren. Sie ist aus dem Emscripten-Compiler entstanden, was zuweilen zu Verwechslungen zwischen der Toolchain und dem Compiler führt [Zak15]. Die Toolchain wird seit 2015 kontinuierlich verbessert, was häufige Wechseln der Tools innerhalb der Toolchain bedingt [Zak16]. Die Emscripten-Toolchain bietet im Hinblick auf die Einbindung von WebAssembly in eine Website einige Vorteile. Emscripten vereinfacht u.a. den Zugriff auf Methoden in WebAssembly, die Einbindung

von WebAssembly in eine Website und emuliert, falls nötig, Features, die nicht in der Browser-Umgebung zur Verfügung stehen. Das hat jedoch den Nachteil, dass ein Overhead entsteht, der gegebenenfalls nicht nötig ist. Emscripten erzielt diese Vorteile, indem neben einer Datei mit WebAssembly auch eine HTML-Datei mit JavaScript-Code ausgegeben wird. Der JS-Code wird als Glue Code bezeichnet, da er den WebAssembly-Code an den JS-Code „bindet“ [Ems20e].

Abbildung 7: Emscripten-Logo

Abbildung 8 zeigt die Emscripten-Toolchain. Auffällig ist, dass sie zum Teil dieselben Tools wie die Clang-LLVM-Toolchain nutzt. Sie unterscheiden sich nur im Compiler-Frontend und in einem zusätzlichen Tools hinter dem Backend. Im Frontend wird anstelle von Clang emcc genutzt. Nach dem Backend wird der Wasm-Code mithilfe von Binaryen weiter optimiert. Die IR (LLVMIR), der Code-Optimierer (LLVM) und das Backend (LLVM WebAssembly Backend) sind identisch.

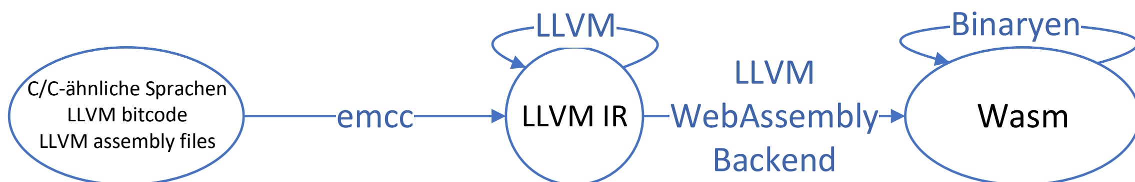


Abbildung 8: Emscripten-Toolchain

emcc ist ein Compiler-Frontend, das speziell für die Emscripten-Toolchain entwickelt wurde. Es ist, wie Clang, ein Drop-in Replacement für GCC. Das erleichtert das Ersetzen eines vorhandenen Frontends in einem Projekt durch emcc. Im Vergleich zu Clang kann emcc mehr Sprachen verarbeiten. Neben C- und C-ähnlichen Sprachen können LLVM-Bitcode und LLVM-Assembler-Code verarbeitet werden [Ems20b].

Binaryen ist ein Compiler und eine Toolchain mit eigener IR, der sogenannten Binaryen IR. Das Tool zielt auf die Optimierung von WebAssembly ab. Der Hauptunterschied zwischen Binaryen und LLVM ist, dass Binaryen Wasm als Eingabe nutzt. Bereits zu WebAssembly kompilierter Code wird optimiert und wieder als WebAssembly-Code ausgegeben. Binaryen kann so in andere Toolchains eingebettet werden. Das Ziel ist eine weitere Optimierung des Codes. Binaryen kann WebAssembly stärker optimieren als LLVM, da es nur auf eine Zielsprache, nämlich WebAssembly, ausgelegt ist [Zak18].

3 Evaluierung der Geschwindigkeit von WebAssembly

Lade- und Ausführungsgeschwindigkeiten sind entscheidende Faktoren für die Usability einer Website. Hohe Geschwindigkeiten steigern die Usability, geringe senken sie. Im schlechtesten Fall verlassen Besucher eine Website sogar, weil sie zu langsam ist. Laut einer Studie von Akamai brechen 53% der potenziellen Besucher einer Website einen Seitenaufruf ab, wenn die Seite nicht innerhalb von drei Sekunden geladen ist [Aka17]. Wurde die Seite geladen, so rückt die Ausführungsgeschwindigkeit in den Fokus. Entspricht sie nicht den Erwartungen der Besucher, so tendieren sie dazu, die Seite zu verlassen und eine andere aufzusuchen. Lade- und Ausführungsgeschwindigkeiten sollten daher so hoch wie möglich sein, um Nutzer auf der eigenen Webseite zu halten.

WebAssembly verspricht eine Steigerung der Geschwindigkeit, insbesondere der Ausführungsgeschwindigkeit. Laut einer Veröffentlichung der Entwickler von WebAssembly aus 2017 soll Wasm 33,7% schneller als asm.js (optimierter, typsicherer JS-Code) und nur 10% langsamer als nativer Code sein [Haa+17]. Dieses Versprechen wurde bereits in diversen Benchmarks überprüft. Die erzielten Ergebnisse unterscheiden sich jedoch stark. Auf der einen Seite ist von Geschwindigkeitsvorteilen im Bereich des 20-fachen der Geschwindigkeit von JavaScript die Rede [Abo19a]. Auf der anderen Seite wird von Geschwindigkeitsnachteilen im Bereich des 300-fachen gesprochen [Bli18].

Die vorhandenen Benchmarks lassen keinen einheitlichen Rückschluss auf die Geschwindigkeit von WebAssembly zu. Daher werden in einer theoretischen Untersuchung die Geschwindigkeitsvor- und -nachteile von Wasm aufgrund seiner Architektur untersucht. Dabei zeigt sich, wo WebAssembly im Vergleich zu JS Vor- und Nachteile hat. Der Vergleich mit JS wird durchgeführt, da JS die aktuell am weitesten verbreitete Technologie zur clientseitigen Ausführung von Code auf Webseiten ist. Nach der theoretischen Untersuchung werden die gewonnenen Ergebnisse in einer praktischen Untersuchung überprüft.

In der praktischen Untersuchung wird die Geschwindigkeit von WebAssembly im Vergleich zu JavaScript und C gemessen. Die Geschwindigkeit wird dabei in Ladegeschwindigkeit und Ausführungsgeschwindigkeit aufgeteilt. Die Ladegeschwindigkeit ist die Geschwindigkeit, in der eine Website geladen wird. Sie wird gemessen, in dem die Zeit vom Seitenaufruf bis zu dem Zeitpunkt erfasst wird, in dem die Seite verwendet werden kann. Die Ausführungsgeschwindigkeit ist die Geschwindigkeit, in der eine Seite Code ausführt. Sie wird bestimmt, indem die Zeit zur Ausführung einer rechenintensiven Aufgabe gemessen wird.

Nach der praktischen Untersuchung wird in einem Fazit die Auswirkung der theoretischen und praktischen Untersuchung auf produktiv eingesetzte Anwendungen dargestellt. Darauf aufbauend wird eine Empfehlung gegeben, ob WebAssembly aufgrund seiner Geschwindigkeit verwendet werden sollte oder nicht.

3.1 Theoretische Untersuchung der Geschwindigkeit

WebAssembly bietet im Vergleich zu JavaScript in einigen Bereichen Geschwindigkeitsvorteile, in anderen Bereichen jedoch auch -nachteile. Um zu erkennen, woher die Vor- und Nachteile stammen hilft es zu wissen, woraus sich die Laufzeiten von WebAssembly- und JavaScript-Anwendungen zusammensetzen. Die Zusammensetzung der Laufzeiten wird daher im Folgenden gezeigt.

JS- und Wasm-Anwendungen werden beide in der JavaScript Virtual Machine (JSVM) des Browsers ausgeführt. Die Aufgabe der JSVM besteht darin, den Quellcode der Anwendung in Maschinencode zu übersetzen und auszuführen. Der pure Quellcode/ WebAssembly-Bytecode kann nicht ausgeführt werden. Erst der Maschinencode kann ausgeführt werden. Bei der Übersetzung ergeben sich Unterschiede in der Laufzeit, da die JSVM den JS- und Wasm-Code unterschiedlich behandelt. Die unterschiedliche Behandlung ist damit begründet, dass JS-Code Hochsprachencode und WebAssembly-Code Bytecode ist. Der WebAssembly-Code ist näher am Maschinencode, sodass weniger Schritte erforderlich sind, um ihn in Maschinencode zu übersetzen [Cla17f].

Mehrere Schritte haben Einfluss auf die Laufzeit von JS-Code. Abbildung 9 listet die Schritte auf.

Abbildung 9: Schritte, die Einfluss auf die Laufzeit von JS-Code haben

Bevor mit der Verarbeitung des JS-Codes begonnen werden kann, muss er heruntergeladen werden. Daraufhin wird er von einem Parser in eine Form gebracht, die der JIT-Compiler der JSVM verarbeiten kann. Anschließend interpretiert, kompiliert und optimiert der JIT-Compiler den Code. Da JS typunsicher ist, müssen beim Optimieren Annahmen über den Code, beispielsweise über Datentypen, getroffen werden. Diese können sich als falsch herausstellen. Ist das der Fall, muss der Code reoptimiert werden. Anschließend kann der Code ausgeführt und der benutzte Speicher vom Garbage Collector freigegeben werden [Cla17f].

Wichtig zu erwähnen ist, dass die einzelnen Schritte beim Ausführen von JS-Code nicht nur einmal durchlaufen werden. Es gibt einen zyklischen Ablauf, da immer nur Teile des Codes verarbeitet werden. Es wird etwas Code geparst, dann wird etwas kompiliert und dann ausgeführt, dann wird wieder geparst, wieder kompiliert und wieder ausgeführt und so weiter [Cla17f]. Manche Schritte finden parallel statt. So wird Code beim ersten Durchlauf JIT-kompiliert. Gleichzeitig wird der Code von einem weiteren Compiler optimiert. Sobald die Optimierung durchgeführt wurde, wird der optimierte Code anstelle des JIT-kompilierten Codes genutzt [Cla17c].

Auf die Laufzeit von WebAssembly-Code haben nur vier Schritte Einfluss. Der Code muss heruntergeladen, dekodiert, kompiliert und optimiert, sowie ausgeführt werden. Der Schritt „Parsen“ wird durch den Schritt „Dekodieren“ ersetzt. Es werden dabei verschiedene Aufgaben ausgeführt. Das Ergebnis ist jedoch dasselbe, nämlich Bytecode für die JSVM. Die Schritte „Re-Optimieren“ und „Garbage Collection“ entfallen. Abbildung 10 zeigt die Schritte, die Einfluss auf die Laufzeit von WebAssembly nehmen.

Abbildung 10: Schritte, die Einfluss auf die Laufzeit von Wasm-Code haben

Begonnen wird bei Wasm, wie auch bei JS, mit dem Download des Codes. Daraufhin wird er dekodiert und an den Compiler übergeben. Dieser kompiliert und optimiert ihn. Abschließend wird er ausgeführt [Cla17f].

Beim Vergleich der Schritte für JS-Code und Wasm-Code stellen sich zwei Fragen: Warum gibt es bei Wasm nur vier Schritte und welchen Einfluss hat die Verringerung der Schritte auf die Laufzeit des Codes? Beide Fragen werden in den folgenden Unterkapiteln beantwortet. Jedes Unterkapitel befasst sich mit einem Schritt. Darin wird auf die Unterschiede und Gemeinsamkeiten zwischen JS und Wasm eingegangen. Daraus ergibt sich eine Erwartung an die Laufzeit von WebAssembly-Code.

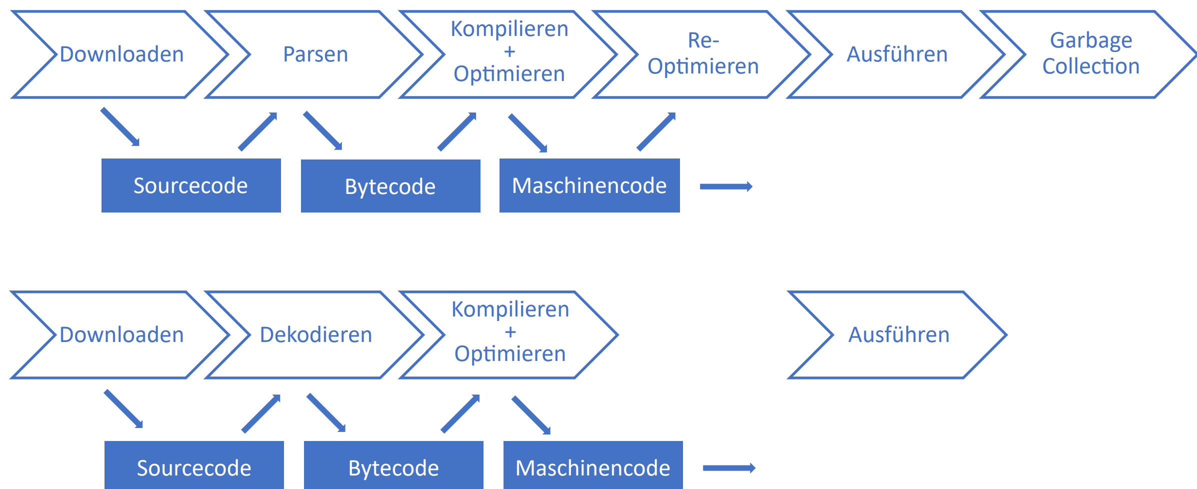
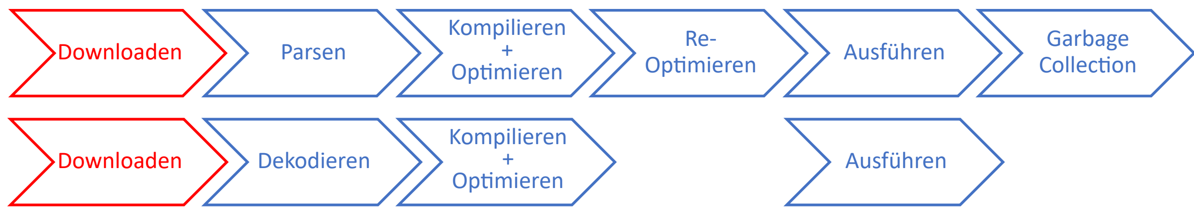


Abbildung 11: Vergleich der Ausführungsschritte von JS und Wasm

3.1.1 Download-Zeiten



Bevor JS- oder Wasm-Code verarbeitet werden kann, muss er heruntergeladen werden. Der Download des Codes ist damit der erste Schritt, der die Laufzeit beeinflusst. Die benötigte Zeit zum Download hängt maßgeblich von zwei Faktoren ab. Zum einen von der Netzwerkgeschwindigkeit und zum anderen von der Codemenge. Auf die Netzwerkgeschwindigkeit haben JS und Wasm keinen Einfluss, auf die Codemenge jedoch schon. Weniger Code bedeutet kleinere Dateien. Kleinere Dateien bedeuten kürzere Downloadzeiten. Die Verringerung der Dateigrößen hat somit direkten Einfluss auf die Ladezeit einer Website. Es ist daher vorteilhaft, kleine Dateien zu nutzen, um die Ladezeit einer Website gering zu halten.

JS-Code ist Hochsprachencode. Er enthält viele Zeichen, die keinen Einfluss auf die Funktion des Codes haben. Beispiele dafür sind lange Variablennamen, Kommentare, Leerzeichen und Zeilenumbrüche. Sie erleichtern das Verständnis des Codes, erhöhen jedoch die Dateigröße. Zusätzlich kann JS-Code nicht genutzte Teile enthalten, beispielsweise ungenutzte Funktionen. Die Dateigröße kann jedoch durch JS-Minifier verringert werden. Sie entfernen alle Zeichen, die keinen Einfluss auf die Funktionalität des Codes haben.

WebAssembly ist Bytecode, der von einem Compiler erzeugt wird. Der Compiler optimiert den Code je nach gesetztem Compiler-Flag. Bei einer Optimierung mit dem Flag „-O2“ oder „-O3“, wie sie in vielen Projekten stattfindet, reduziert sich die Größe des erzeugten Codes. Der Compiler erreicht die Reduktion, indem er unter anderem alle Zeichen entfernt, die keinen Einfluss auf die Funktion des Codes haben. Zudem wird nicht genutzter Code eliminiert. Bei der Kompilierung mit der Emscripten- und Clang-LLVM-Toolchain werden weitere Optimierungen mit Einfluss auf die Dateigröße vorgenommen. Sie können auf [Cla20b] nachgelesen werden.

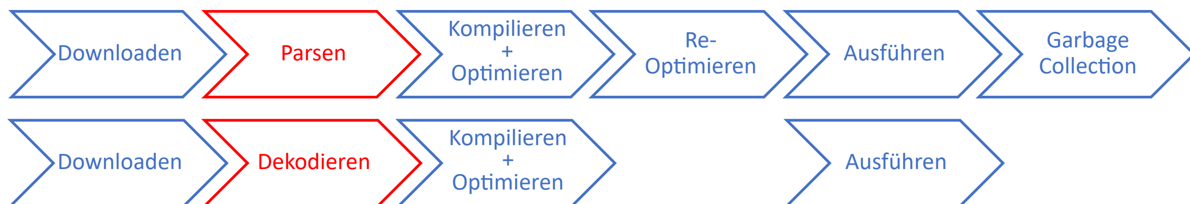
Eine weitere Reduzierung der Größe von .wasm-Dateien im Vergleich zu .js-Dateien ergibt sich aus der Kodierung der Dateien. .wasm-Dateien nutzen das Binärformat, .js-Dateien das Textformat. Experimente zeigen, dass aufgrund dieses Unterschieds bei vergleichbarem Inhalt .wasm-Dateien durchschnittlich 20-30% kleiner sind als .js-Dateien [W3C20a].

Bei WebAssembly kommt zur Größe des Bytecodes noch eine gewisse Menge an Glue Code hinzu. Er muss zusätzlich zum Bytecode heruntergeladen werden. Das beeinflusst die Ladezeit einer Website negativ. Der Glue Code wird benötigt, um den Wasm-Code zu laden und zusätzliche Funktionen zur Verfügung zu stellen. Die Menge an Glue Code unterscheidet sich in Abhängigkeit von der Funktionalität des Codes und der verwendeten Toolchain. Alle Funktionen, die nicht nativ im Browser zur Verfügung stehen, müssen darin nachgebildet werden. Nutzt der Code zum Beispiel das Dateisystem, so muss der Glue Code ein Dateisystem im Browser nachbilden.

Die Clang-LLVM-Toolchain erzeugt keinen Glue Code. Der Entwickler muss diesen selbst schreiben. Der Glue Code wird daher wahrscheinlich auf den Bytecode zugeschnitten sein. Die Emscripten-Toolchain liefert den Glue Code mit. Er ist grob auf den Bytecode zugeschnitten. Trotzdem enthält er nicht benötigte Funktionen. Sie stellen Overhead dar, der die Ladezeit einer Website unnötig erhöht.

Aus den Erkenntnissen zu den Größen von .js- und .wasm-Dateien ergibt sich die Erwartung für die praktische Untersuchung, dass bei gleicher Funktionalität WebAssembly-Code eine geringere Größe hat als JavaScript-Code. Da bei WebAssembly jedoch der Glue Code hinzukommt, bleibt abzuwarten, ob der Größenvorteil des Codes nicht kompensiert wird.

3.1.2 Parsing- / Dekodierungszeiten



Nachdem der Code heruntergeladen wurde, muss er in eine Form umgewandelt werden, mit der der Compiler der JSVM arbeiten kann. Die Form wird als Bytecode bezeichnet. Sie ist eine Intermediate Representation (IR) für die JSVM. Für JS-Code wird die Umwandlung „Parsen“ genannt, für Wasm-Code „Dekodieren“.

JavaScript-Code wird vom Parser in einen Abstract Syntax Tree (AST) umgewandelt (engl. geparst). Ein AST ist eine hierarchische Darstellung der Syntax des Codes. Vom AST aus wird der Code in Bytecode für die JSVM konvertiert [Cla17f].

WebAssembly-Code ist bereits Bytecode. Er muss daher nicht in einen AST und weiter in Bytecode konvertiert werden. Er muss nur dekodiert werden. Dekodieren ist einfacher

und schneller als Parsen [Cla18b]. Der Hauptgrund dafür ist, dass Variablen- und Funktionsnamen in Wasm als Indizes und nicht wie in JS, als Strings dargestellt werden. Die Darstellung über Indizes erlaubt das Auffinden der Funktionen über ihren Index. Das ist schneller als ein Dictionary-Lookup, der bei der Verwendung von Strings durchgeführt werden muss [W3C20a].

.wasm-Dateien sind binär kodiert. Daher ist vor der Kompilierung eine Dekodierung in Bytecode erforderlich [Cla17f].

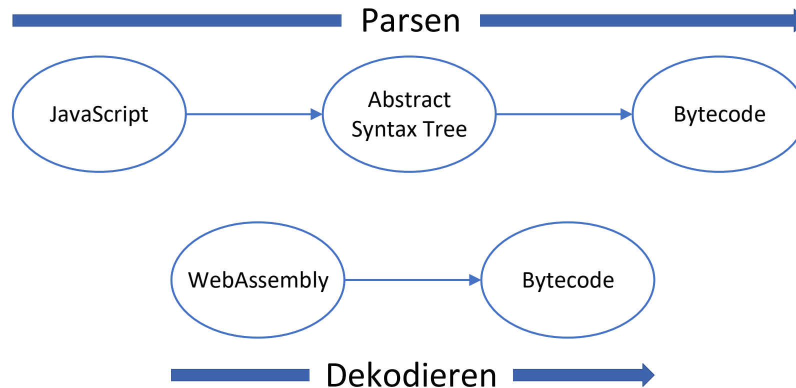


Abbildung 12: Vergleich Parsen und Dekodieren

Experimente zeigen, dass Dekodieren ca. um den Faktor 23 schneller ist als Parsen von vergleichbarem JS-Code [W3C20a].

Ein weiterer Geschwindigkeitsvorteil wird durch die Aufteilung der Dekodierung auf mehrere Threads erreicht. JS-Code wird von nur einem Thread geparst. Wasm-Code wird von mehreren Threads parallel dekodiert. Dadurch ist die Dekodierung schneller beendet und es kann früher mit der Kompilierung begonnen werden. Das reduziert die Ladezeit einer Website [Cla18b].

Abbildung 13 zeigt das Parsen von JS-Code in einem Thread. Es ist zu erkennen, dass nachdem das letzte Paket angekommen ist noch eine gewisse Zeit lang geparst wird. Erst nach dem Ende des Parsens kann mit der Kompilierung begonnen werden. Im Vergleich dazu ist in Abbildung 14 das Dekodieren von Wasm-Code dargestellt. Es wird deutlich, dass jedes Paket in mehreren Threads parallel dekodiert wird. Dadurch ist die Zeit, die nach dem Ankommen des letzten Paketes benötigt wird, geringer als bei JS und es kann schneller mit dem Kompilieren begonnen werden.

Für die praktische Untersuchung ergibt sich daraus, dass eine WebAssembly-Anwendung vermutlich schneller geladen werden kann als eine vergleichbare JavaScript-Anwendung.

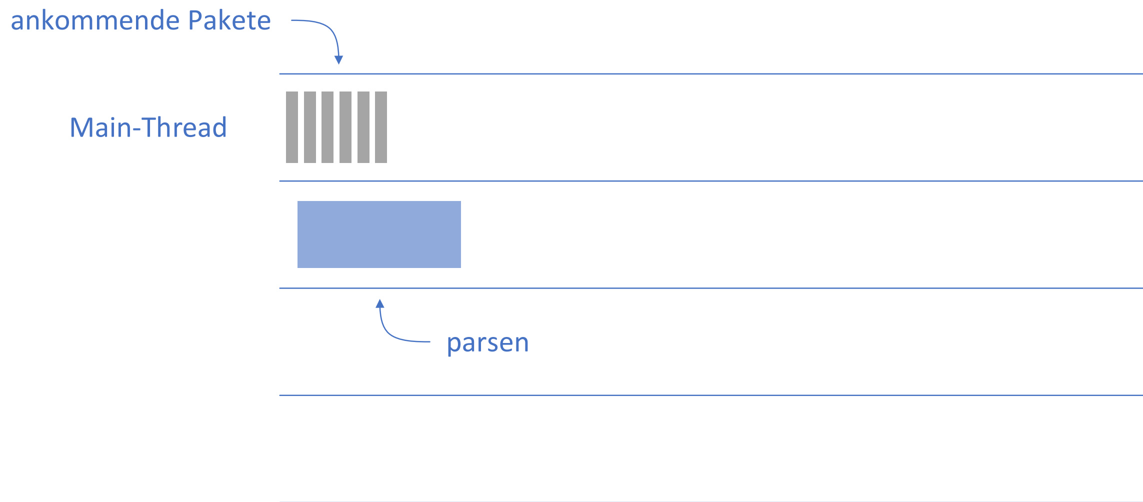


Abbildung 13: Gantt-Diagramm der JS-Ausführungszeit Teil 1: Parsen [Cla18b]

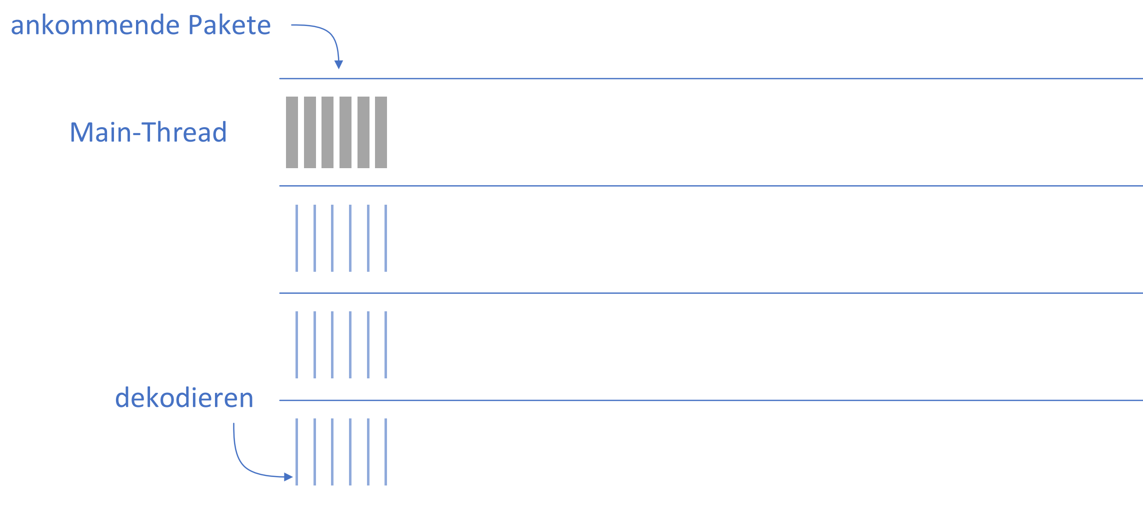
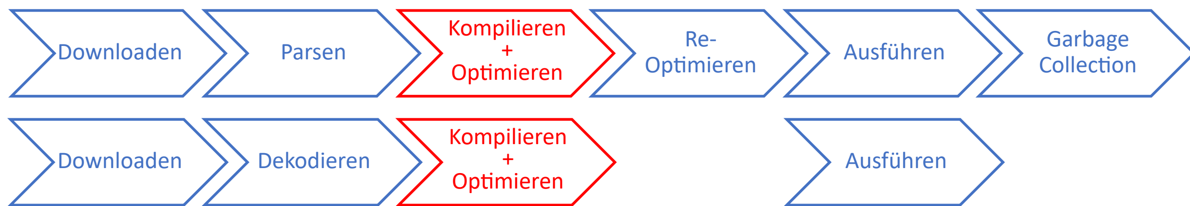


Abbildung 14: Gantt-Diagramm der Wasm-Ausführungszeit Teil 1: Parsen [Cla18b]

3.1.3 Kompilierungs- und Optimierungszeiten



Bei der Kompilierung wird aus dem beim Parsen/Dekodieren gewonnen Bytecode Maschinencode erzeugt. Erst der Maschinencode kann auf dem System des Nutzers ausgeführt werden. Die zum Kompilieren benötigte Zeit trägt zur Ladezeit einer Webseite bei. Auf die Kompilierung folgt die Optimierung des erzeugten Maschinencodes. Sie findet statt, um die Performance des Maschinencodes zu steigern. Die Performance ist nach erstmaliger Kompilierung aufgrund eines Trade-offs gering und sollte für hohe Ausführungsgeschwindigkeiten gesteigert werden. Die Optimierung benötigt jedoch Rechenzeit, wodurch die Ausführungsgeschwindigkeit einer Website vorerst sinkt. Der optimierte Code kann jedoch schneller ausgeführt werden, sodass die Ausführungsgeschwindigkeit einer Seite wieder steigert.

Bei der Kompilierung und Optimierung von Bytecode aus JS- und Wasm-Code gibt es Unterschiede. Sie entstehen, da die JSVM den JS-JSVM- und Wasm-JSVM-Bytecode unterschiedlich behandelt. Daraus ergeben sich unterschiedliche Lade- und Ausführungsgeschwindigkeiten. Die verschiedenen Verfahren zur Kompilierung und Optimierung werden im Folgenden betrachtet, um Rückschlüsse auf die Geschwindigkeiten von Anwendungen zu ziehen [Cla18b].

Die Kompilierung von JS-Code erfolgt lazy zur Laufzeit. Lazy bedeutet, dass Code erst in dem Moment kompiliert wird, in dem er benötigt wird. Das reduziert den Einfluss der Kompilierungszeit auf die Ladezeit einer Seite. Sobald Code benötigt wird, wird er von einem JIT-Compiler in Maschinencode kompiliert und ausgeführt [Cla17c]. Abbildung 15 zeigt das JIT-Kompilieren von Code zur Laufzeit.

Neben dem Vorteil des geringeren Einflusses auf die Ladezeit einer Website hat Lazy-Kompilierung jedoch auch Nachteile. Da der Code erst kompiliert wird, wenn er benötigt wird, kommt zur Ausführungszeit die Kompilierungszeit des Codes hinzu. Das senkt die Ausführungsgeschwindigkeit einer Website. Weiterhin muss bei der JIT-Kompilierung die Kompilierungsgeschwindigkeit hoch sein. Je höher sie ist, desto höher ist die Ausführungsgeschwindigkeit einer Seite. Beim Kompilieren gibt es einen Trade-off. Es muss zwischen der benötigten Zeit zum Kompilieren und der Geschwindigkeit des erzeugten Maschinencodes abgewogen werden. Je mehr Zeit in die Kompilierung investiert wird, desto schneller ist der erzeugte Code. Mit längeren Kompilierungszeiten sinkt jedoch die Ausführungsgeschwindigkeit einer Seite vorerst [Cla17f].

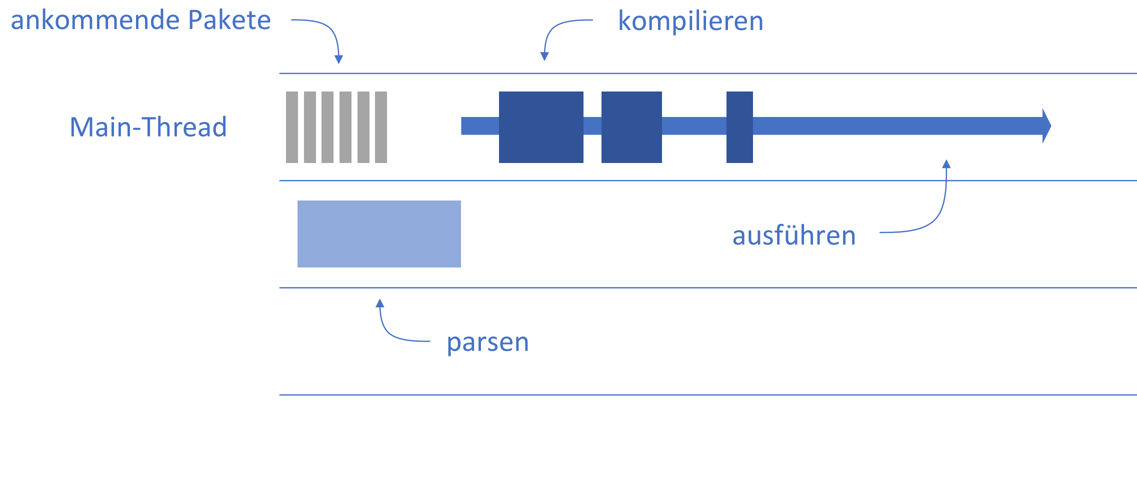


Abbildung 15: Gantt-Diagramm der JS-Ausführungszeit Teil 2: Kompilieren [Cla18b]

Die JSVM löst den Trade-off zwischen Kompilierungszeit und Optimalität des Maschinencodes, indem sie beobachtet, wie häufig Codeteile ausgeführt werden. Die Beobachtung findet durch den „Monitor“ statt. Wird ein Codeteil selten ausgeführt so wird die Kompilierungsgeschwindigkeit priorisiert. Die Optimalität des Maschinencodes wird vernachlässigt. Wird ein Codeteil häufig ausgeführt, so wird die Optimalität des Maschinencodes priorisiert. Es wird viel Zeit in die Kompilierung investiert. Der erzeugte Maschinencode für diesen Codeteil läuft dadurch performant. So versucht der Monitor die Geschwindigkeit einer Website hoch zu halten. Die Erzeugung von performantem Maschinencode wird als Optimierung bezeichnet. Sie wird im Folgenden Kapitel untersucht [Cla17c].

Die Kompilierung von Wasm-Code findet nicht wie bei JS lazy statt, sondern bereits bevor mit der Ausführung begonnen wird. Abbildung 16 zeigt, dass der Code komplett vor der Ausführung in Maschinencode kompiliert wird. Das hat Vorteile und Nachteile. Vorteilhaft ist, dass zur Laufzeit keine Zeit zum Kompilieren benötigt wird. Nachteilig ist, dass der komplette Code in Maschinencode kompiliert werden muss, bevor die erste Zeile Code ausgeführt werden kann. Durch die komplette Kompilierung zu Beginn ist die Ausführungsgeschwindigkeit von Wasm höher als die von JS. Um den dadurch entstandenen Nachteil bei der Ladezeit auszugleichen nutzt Wasm ein Verfahren namens Streaming Compilation. Es reduziert die Ladezeit so drastisch, das kurz nach dem Downloaden und Dekodieren des Codes mit der Ausführung begonnen werden kann [Cla18b].

Streaming Compilation ist ein Verfahren, bei dem in mehreren Threads gleichzeitig Code dekodiert und kompiliert wird. Dieses Verfahren reduziert die Ladezeit stark. Streaming Compilation zeichnet aus, dass beim Laden von Wasm-Code nicht gewartet wird, bis der Download abgeschlossen ist. Sobald ein Paket der .wasm-Datei heruntergeladen wurde,

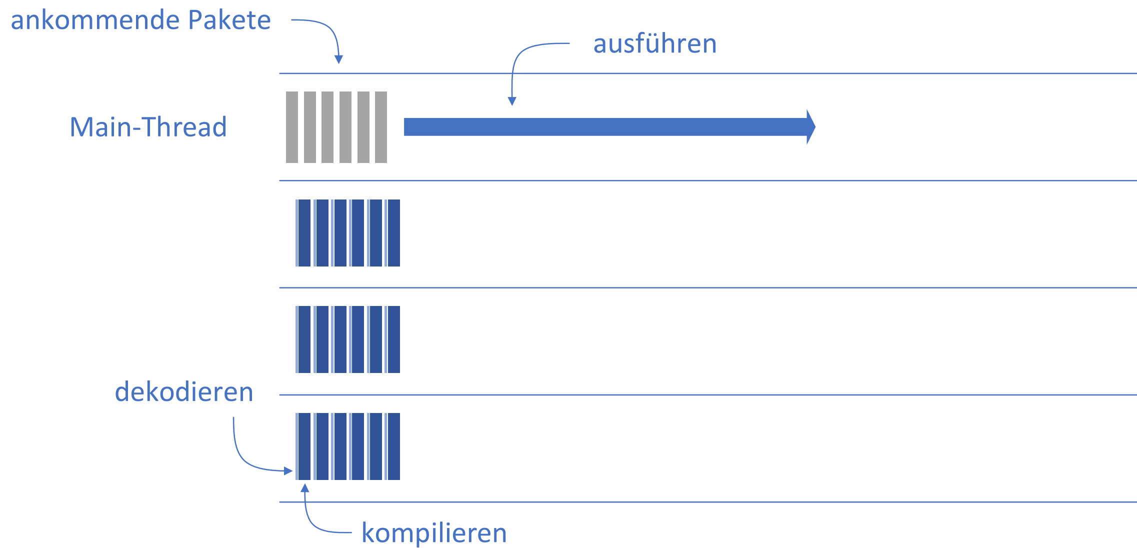


Abbildung 16: Gantt-Diagramm der Wasm-Ausführungszeit Teil 2: Kompilieren [Cla18b]

wird es an einen anderen Thread weitergegeben und dekodiert. Die JSVM arbeitet dabei parallel und kann alle verfügbaren Kerne eines Systems nutzen. Sobald das Paket geparst wurde, wird mit der Kompilierung begonnen. Sie findet auch parallel statt. Auf diese Weise können 30-60 Megabyte WebAssembly-Code pro Sekunde kompiliert werden [Cla18b]. Das Kompilieren ist somit schneller als Code bei einer durchschnittlichen Internetleitung heruntergeladen werden kann [Ook20]. Für die Ladezeit einer Seite bedeutet das, dass das Kompilieren von WebAssembly-Code kaum Einfluss auf sie hat.

Bei der Kompilierung von Wasm-Code in Maschinencode gibt es, wie bei der Kompilierung von JS-Code, einen Trade-off. Es muss zwischen hoher Kompilierungsgeschwindigkeit und Schnelligkeit des erzeugten Codes abgewogen werden. Bei WebAssembly wird der Fokus zuerst auf eine hohe Kompilierungsgeschwindigkeit gelegt, um die Ladezeit gering zu halten. Sobald der komplette Code kompiliert wurde, beginnt die JSVM den Code, parallel zur Ausführung, zu optimieren. Da die Optimierung parallel stattfindet, beeinflusst sie die Ausführungsgeschwindigkeit des Codes nicht. Sobald der Code optimiert wurde, ersetzt er den zu Beginn kompilierten Code. Das Ersetzen des Codes ohne Unterbrechung wird als Hot-Swapping bezeichnet. Dieses Vorgehen beschleunigt die Ausführung des Codes um das Doppelte [Cla18b]. Abbildung 17 zeigt das Optimieren und Hot-Swappen des Codes.

In Zukunft soll bereits kompilierter Wasm-Code gecacht werden. Bei einem Neuladen der Seite muss der Code dann nicht erneut kompiliert werden. Er muss nur noch aus dem Cache geladen werden und ausgeführt werden. Das Cachen des kompilierten Codes beschleunigt die Ladegeschwindigkeit. Dieses Feature existierte bereits 2018 in Firefox.

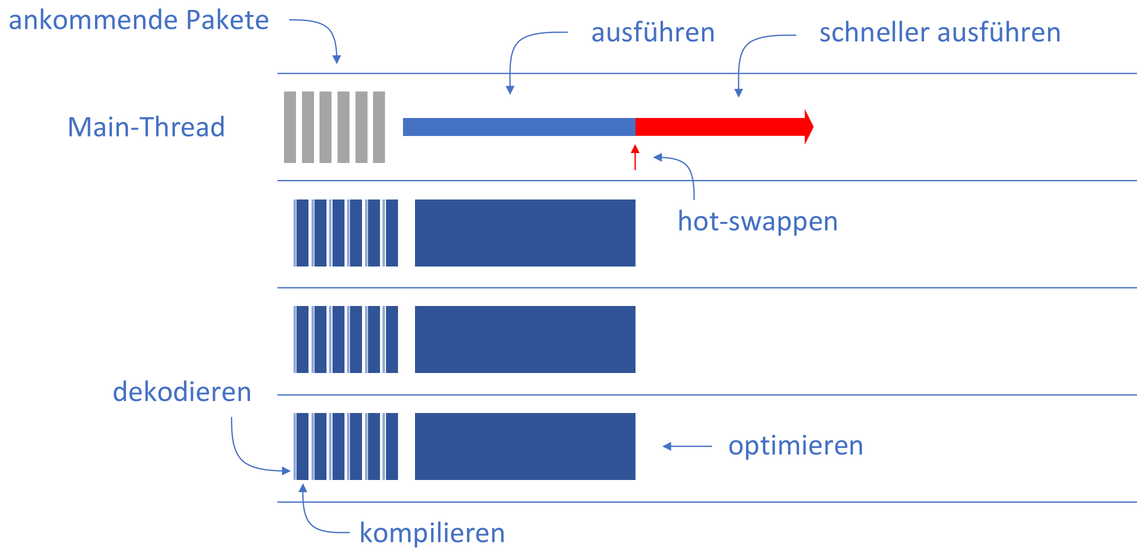
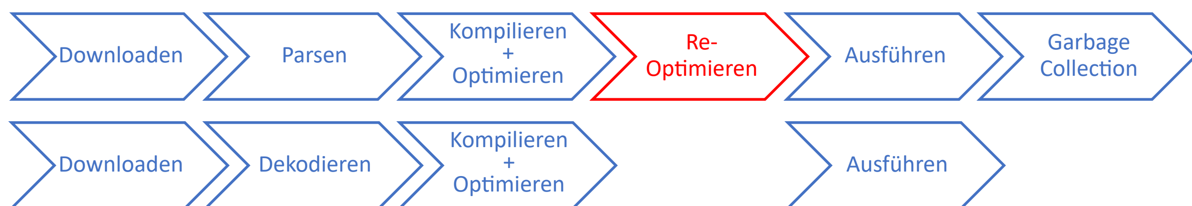


Abbildung 17: Gantt-Diagramm der Wasm-Ausführungszeit Teil 3: Optimieren [Cla18b]

Es wurde aber im selben Jahr wieder entfernt, da es keinen Konsens gab, wie und wo der kompilierte Code effizient gecacht werden kann. Das Feature soll in Zukunft wieder aktiviert werden [Smi18].

Für die praktische Untersuchung ist zu erwarten, dass die Ladegeschwindigkeit von Wasm-Code höher ist als die von JS-Code. Die Ladezeit von Wasm-Code ist wahrscheinlich geringer als die von JS-Code. Das ist bemerkenswert, da der komplette Wasm-Code vor der Ausführung kompiliert wird. Für die Ausführungsgeschwindigkeit ist zu erwarten, dass WebAssembly-Code nach der Optimierung deutlich schneller läuft als JS-Code. Da die Optimierung von Wasm-Code nicht nur bei häufig verwendetem Code stattfindet, sondern immer, ist eine gleichmäßige Performancesteigerung zu erwarten. Die Ladezeit wird in Zukunft weiter, durch das Cachen von kompiliertem Code, gesenkt werden.

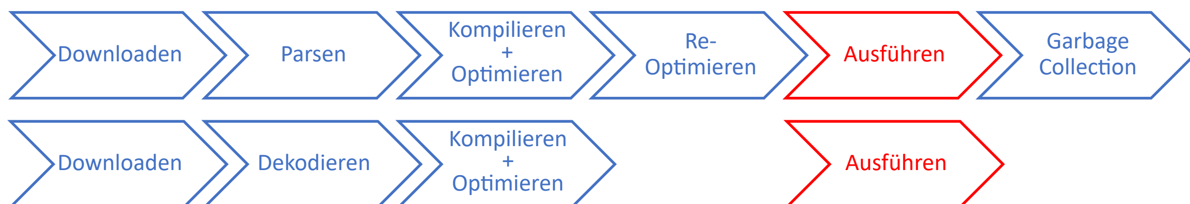
3.1.4 Re-Optimierungszeiten



JS ist eine typunsichere Sprache. Typunsicher bedeutet, dass vor der Ausführung des Codes nicht feststeht, welchen Typ Variablen haben. Dadurch entstehen Unsicherheiten, die die Optimierung des Codes durch den Compiler erschweren. Dieser kann den Code nur effektiv optimieren, wenn er die Typen im Voraus kennt. Der Compiler triff daher Annahmen über den Code. Folgendes Beispiel zeigt eine mögliche Annahme: In einem Array mit 100 Feldern, in dem die ersten 50 Felder Zahlen beinhalten, würde er vermuten, dass in den nächsten 50 Feldern auch Zahlen gespeichert werden. Der Code würde unter der Annahme optimiert, dass das komplette Array aus Zahlen besteht. Das muss in JS aber nicht der Fall sein. So können in ein Array die ersten 99 Werte Zahlen sein und der 100. Wert ein Text. Der optimierte Code passt dann nicht mehr auf den aktuellen Datentyp. Daher wird vor der Ausführung von optimiertem Code geprüft, ob alle Annahmen erfüllt sind. Ist das der Fall, wird der optimierte Code ausgeführt. Ist das nicht der Fall, so wird die Ausführung gestoppt, der optimierte Code verworfen, zurück zum unoptimierten Code gewechselt und die Ausführung fortgesetzt. Wird der unoptimierte Code wieder häufig ausgeführt, so wird der Monitor den Code wieder an den Optimierer übergeben und das Spiel beginnt von Vorne. Das braucht zum einen Zeit, und zum anderen wird die Laufzeit von JS-Code dadurch inkonstant [Cla17c].

WebAssembly hat als typsichere Sprache keine Probleme mit falschen Annahmen beim Optimieren. Die Typen von Variablen stehen bereits vor der Ausführung fest. Der Schritt „Re-Optimieren“ entfällt dadurch. Das spart zum einen Zeit und zum anderen wird die Laufzeit einer Website dadurch konstanter [Cla17f].

3.1.5 Laufzeiten



Der Begriff Laufzeit ist mehrfach belegt. Einerseits beschreibt er die Zeit, die der Maschinencode zur Ausführung benötigt. Sie wird im Englischen als „execution“ bezeichnet [Wik20a]. Anderseits bezeichnet Laufzeit „auch die Phase der Ausführung eines Programms in einem spezifischen Laufzeitkontext“ [Wik20b]. Sie wird im Englischen als „runtime“ bezeichnet. Die englischen Begriffe sind hier eindeutiger. In diesem Kapitel bezeichnet Laufzeit die „execution“, also die Zeit, die der Maschinencode zur Ausführung benötigt.

Die Compiler für Wasm- und JS-Code erzeugen dieselbe Art von Maschinencode. Ist der erzeugte Maschinencode zufälligerweise derselbe, so wird dieser gleich schnell ausgeführt.

Da der Code zuvor jedoch unterschiedlich kompiliert und optimiert wird, ist der erzeugte Maschinencode häufig verschieden. Daher ergeben sich bei der Laufzeit Unterschiede zwischen Wasm- und JS-Code [Cla17f].

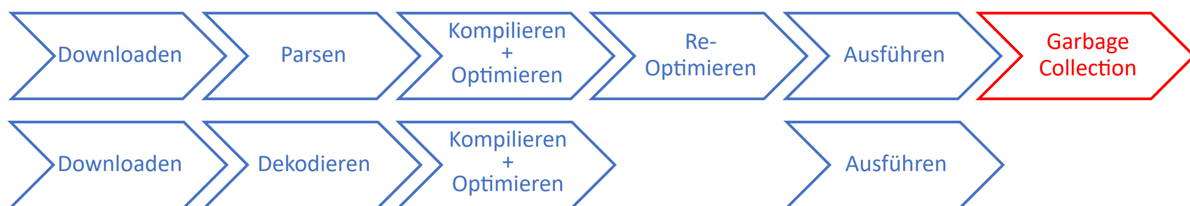
In diesem Schritt zahlt sich bei WebAssembly die mehrfache Optimierung des Codes aus. Die erste Optimierung fand bei der Kompilierung von C-Code in WebAssembly-Code durch LLVM statt. Daran anschließend folgt optional die Optimierung durch Binaryen. Zuletzt wurde der Code durch die JSVM während der Kompilierung in Maschinencode optimiert. Wasm-Code läuft daher tendenziell schneller als JS-Code, bei dem nur einmal optimiert wird. Das ist jedoch nur einer von zwei Gründen für geringere Laufzeiten von Wasm-Code [Cla18b].

Einen weiteren Geschwindigkeitsvorteil gegenüber JS erreicht WebAssembly bei der Ausführung, da immer der komplette Code optimiert wird. JS optimiert nur häufig verwendete Codeteile. In Wasm laufen selten verwendete Codeteile daher schneller als in JS. Das führt zu einer geringeren Laufzeit als bei JS-Code [Cla18b].

Für die Zukunft sind weitere geschwindigkeitssteigernde Features geplant. So steht die Einführung von Atomics und die Implementierung von SIMD auf der Roadmap von WebAssembly [W3C20d].

Zusammenfassend ist für die praktische Untersuchung zu erwarten, das Wasm-Code schneller laufen wird als vergleichbarer JS-Code. Dieser Unterschied wird in Zukunft wahrscheinlich größer werden, wenn die geschwindigkeitssteigernden Features aus der Roadmap implementiert sind.

3.1.6 Garbage Collection-Zeiten



Garbage Collection, zu deutsch automatische Speicherbereinigung, ist bei Programmiersprachen mit automatischem Speichermanagement notwendig. Sie entfernt nicht mehr verwendete Objekte aus dem Speicher und gibt so benutzten Speicher wieder frei. Automatisches Speichermanagement entlastet den Programmierer vom manuellen Umgang mit dem Speicher. Das vermeidet speicherbezogene Bugs und Memory Leaks. Dieser Vorteil wird jedoch mit einem erhöhten Ressourcenbedarf sowie verringerter und unvorhersehbarer Ausführungsgeschwindigkeit erkauft.

JS ist eine Sprache mit automatischem Speichermanagement. Sie benötigt daher einen Garbage Collector. Sobald der Garbage Collector beginnt den Speicher zu bereinigen, wird die Ausführung des JS-Codes unterbrochen. Das reduziert die Ausführungsgeschwindigkeit. Gleichzeitig führt es zu inkonstanten Ausführungszeiten, da nicht bekannt ist, wann der Speicher bereinigt wird.

In WebAssembly muss der Speicher manuell gemanaged werden. Daher wird kein Garbage Collector benötigt. Das verringert die Laufzeit und den Ressourcenbedarf und führt zu konstanten Laufzeiten.

Dass der Speicher manuell gemanaged werden muss, hat dazu geführt, dass ursprünglich nur Sprachen mit manuellem Speichermanagement wie C oder C++ zu WebAssembly kompiliert werden konnten. Aktuell können auch Sprachen wie Java mit automatischen Speichermanagement zu Wasm kompiliert werden. Der Garbage Collector wird dabei in JS und Wasm mit implementiert. Das führt jedoch wieder zu den im vorherigen Absatz genannten Nachteilen.

Zusammenfassend ist für die praktische Untersuchung zu erwarten, dass Wasm-Code durch den Wegfall der Garbage Collection schneller und konstanter läuft als JS-Code.

3.2 Praktische Untersuchung der Geschwindigkeit

In der theoretischen Untersuchung der Geschwindigkeit von WebAssembly sind Erwartungen an die Geschwindigkeit von WebAssembly in der Praxis entstanden. Es wird erwartet, dass Webseiten mit WebAssembly schneller laden und laufen als Seiten, die nur JS nutzt. Um die Erwartungen zu überprüfen, werden in diesem Kapitel die Lade- und Ausführungszeiten einer Website, die WebAssembly nutzt, mit denen einer Website, die nur JS nutzt, verglichen. Zusätzlich zum Vergleich zwischen den Webseiten mit und ohne WebAssembly wird eine native C-Implementierung mit derselben Funktionalität zum Vergleich hinzugezogen.

Im Folgenden wird zunächst der Versuchsaufbau und die Funktionsweise der Webseiten, an denen die Lade- und Ausführungszeiten erfasst werden, erklärt. Das ist notwendig, um nachvollziehen zu können, warum gerade diese Websites verwendet werden. Daraufhin wird das Testsystem, auf dem der Browser und die Webseiten laufen, beschrieben. Dabei wird die Leistung des Systems mit anderen in 2020 genutzten Systemen verglichen. Der Vergleich erlaubt es, die in den folgenden Kapiteln gemessenen Zeiten einordnen zu können. Nach der Beschreibung des Testsystems erfolgt die Messung der Lade- und Ausführungszeiten. Basierend auf den dort gewonnenen Ergebnissen werden die Erwartungen aus der theoretischen Untersuchung überprüft.

3.2.1 Versuchsaufbau

Die praktische Untersuchung der Geschwindigkeit findet anhand einer RC4-Stromchiffre statt. Für eine RC4-Stromchiffre sprechen mehrere Gründe. Zunächst ist sie komplexer als ein Hello-World-Programm. Das ist wichtig, um neben den positiven Aspekten von WebAssembly auch negative Aspekte zu finden. Die positiven Aspekte werden in Tutorials im Web an einfachen, kleinen Programmen umfangreich beschrieben. Auf negative Aspekte wird kaum eingegangen. Ein Grund dafür ist, dass Hello-World-Programme nur minimale Features von WebAssembly nutzen. Eine RC4-Chiffre und ihre Interaktion mit der GUI nutzt mehr Features von Wasm. Mit Blick auf CrypTool-Online (CTO) oder CrypTool 3 (CT3) [Con21] im Browser ist eine Verschlüsselung sinnvoll, da sich das Beispielprogramm im selben Kontext befindet. Derselbe Kontext ist für eine bessere Übertragbarkeit der Ergebnisse auf eine reale Anwendung hilfreich. Für RC4 spricht auch die simple Implementierbarkeit und zahlreiche, bereits existierende Implementierungen im Internet.

Für die Geschwindigkeitstests werden die Laufzeiten von vier Implementierungen in unterschiedlichen Sprachen miteinander verglichen. Eine Implementierung ist in C und eine in JavaScript geschrieben. Sie stellen herkömmliche Desktop-/ Webanwendungen dar. Zwei Implementierungen sind in WebAssembly geschrieben. Sie stellen Websites dar, die den neuen Webstandard WebAssembly nutzen. Die beiden Wasm-Implementierungen unterschieden sich darin, dass sie mit verschiedenen Toolchains kompiliert wurden. Eine Implementierung wurde mit der Clang-LLVM-Toolchain kompiliert, die andere mit der Emscripten-Toolchain. Die Unterscheidung zwischen den Toolchains wird gemacht, um herauszufinden, welche Toolchain effizienteren WebAssembly-Code erzeugt.

Die vier Implementierungen haben denselben Programmablauf. Ein Nutzer gibt einen Schlüssel als Hex-Wert und eine Klartextlänge ein. Das Programm konvertiert den Hex-Schlüssel in einen Dec-Schlüssel und speichert ihn ab. Daraufhin erzeugt es einen zufälligen Klartext in der eingegebenen Länge. Bis zu diesem Schritt unterscheiden sich die Anwendungen nur leicht. Im nächsten Schritt sind die Implementierungen jedoch verschieden. Der erzeugte Klartext wird mit dem Dec-Schlüssel verschlüsselt. Die Verschlüsselung findet in JS, Wasm oder C statt. Nachdem sie beendet ist, wird der Anfang des Geheimtextes im Graphical User Interface (GUI) ausgegeben. Mit der Ausgabe des Geheimtextes ist der Programmablauf beendet.

Verschiedene Implementierungen einer Chiffre können unterschiedlich performant sein. Um vergleichbare Ergebnisse zu erhalten, müssen alle Implementierungen gleich oder sehr ähnlich sein. Um die Vergleichbarkeit zu erhalten, wurde auf die Nutzung bereits vorhandener Implementierungen verzichtet. Sie wurden selbst erstellt und sind sich sehr ähnlich.

Bei den Implementierungen der RC4-Chiffre wurde bewusst auf algorithmusspezifische Optimierungen und besonders performante Operationen verzichtet. Dadurch kommen die Implementierung realen Anwendungen nahe, die nur selten stark auf Performance optimiert sind. Es wird beispielsweise auf Bitoperationen verzichtet. Stattdessen werden rechenintensive Multiplikationen und Divisionen genutzt. Zudem werden keine optimierten Datentypen zum Speichern von Schlüssel, Klartext und Geheimtext verwendet. Alle gespeicherten Werte befinden sich in einer Spanne von 0-255. 8 Bit unsigned Integers hätten für die Speicherung ausgereicht. Es werden jedoch herkömmliche 32 Bit Integers verwendet. Das soll Implementierungen realer Anwendungen nachbilden.

Um die korrekte Funktion der Implementierungen sicherzustellen, wurden die Ausgaben anhand von Testvektoren für die RC4-Stromchiffre der Internet Engineering Task Force (IETF) überprüft [Str11]. Die Implementierungen liefern korrekte Ergebnisse. Eine korrekte Implementierung ist grundlegend wichtig für die Vergleichbarkeit der Geschwindigkeiten der Webseiten.

Die Programme sind im Anhang und unter `https://grthor.github.io/` verfügbar. Dort können sie auf dem eigenen PC ausgeführt werden.

In den Geschwindigkeitsvergleichen in den folgenden Unterkapiteln wird jeweils die Wall Clock Time gemessen. Es ist bekannt, dass die Wall Clock Time auf unterschiedlichen PCs und je nach aktueller Last auf dem System verschieden sein kann. Die Vergleiche sollen jedoch keine durchschnittlichen Laufzeiten auf allen PCs ermitteln, sondern die Differenzen zwischen den unterschiedlichen Implementierungen zeigen. Auf eine gleiche Lastsituation auf dem Testsystem wurde während der Vergleiche geachtet.

3.2.2 Testumgebung

Die folgenden Geschwindigkeitsvergleiche (Benchmarks) wurden auf einem PC mit einem AMD Ryzen 5 3600 6-Kern-Prozessor (4200 MHz) und 16 GB DDR4 (2133 MHz) Arbeitsspeicher erstellt. Auf dem System läuft ein aktuelles Windows 10 10.0.19041. Nach einem Benchmark des PCs mit PCMark 10 ist der PC im oberen Fünftel der 2020 mit PCMark 10 gebenchmarkten PCs einzuordnen. Damit ist er besser als der Durchschnitts-PC [UL20]. Für die Benchmarks in den folgenden Kapiteln bedeutet das, dass sie auf vielen PCs langsamer laufen werden.

3.2.3 Ladegeschwindigkeiten

Die theoretische Untersuchung der Geschwindigkeit von Wasm lässt erwarten, dass die Ladegeschwindigkeit von Websites, die Wasm nutzen, höher ist, als die Ladegeschwin-

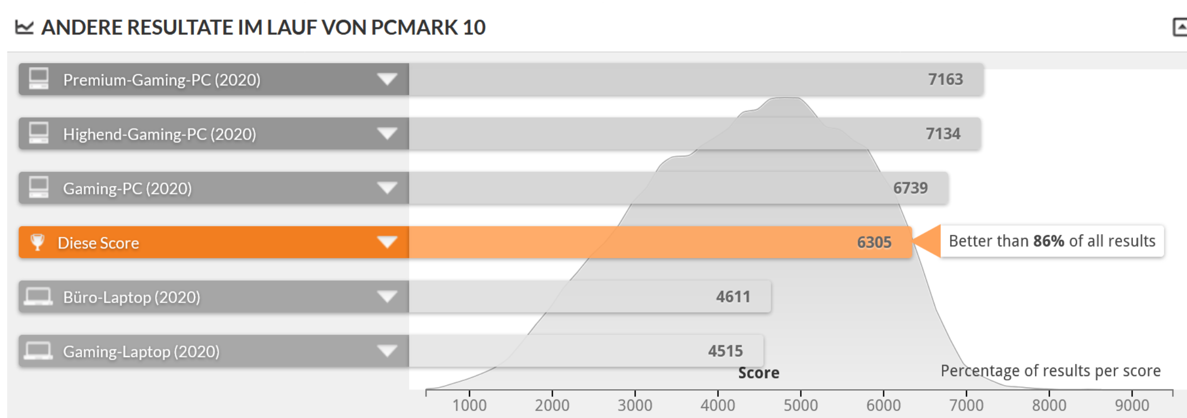


Abbildung 18: Leistung des Testsystems im Vergleich zu anderen Systemen

digkeit von Websites, die JS nutzen. Begründet wurde die Annahme zum einen mit kleineren Dateien bei Wasm und somit kürzeren Downloadzeiten und zum anderen mit der schnelleren Dekodierung und Kompilierung des Codes. Beides soll zu einer geringen Verzögerungen führen, bis der Code ausgeführt werden kann.

Die Ladezeiten der vier RC4-Implementierungen werden in diesem Kapitel gemessen und verglichen. Anhand der Testergebnisse sollen die Annahmen über die Ladegeschwindigkeit aus der theoretischen Untersuchung überprüft werden.

Einen großen Einfluss auf die Ladezeit und somit auf die Ladegeschwindigkeit einer Website hat die Menge an Code, die vor der Nutzung heruntergeladen werden muss. Die verschiedenen Implementierungen unterscheiden sich deutlich in ihrer Größe. Abbildung 19 zeigt die Unterschiede. Die Größe hat nur bei den drei Web-Implementierungen Einfluss auf die Ladezeit und so auf die Usability der Webseite. Bei der C-Implementierung findet der Download einmalig vor der Nutzung statt und beeinflusst die Usability der Anwendung nur wenig. Die Größe der C-Implementierung wird als Vergleichswert jedoch trotzdem angegeben.

Die HTML- und JS-Dateien sind mithilfe von Code-Minifiern minimiert. Dadurch fließt nur der Code in die Messung ein, der Funktionalität mit sich bringt. Kommentare, Zeilenumbrüche und Ähnliches beeinflussen die Messung nicht. Die Nutzung von Code-Minifiern trägt zur Vergleichbarkeit der Ergebnisse bei.

Abbildung 19 lässt erkennen, dass die JS-Implementierung mit 3,9 kB am kleinsten ist. Nur etwas größer ist die Clang-LLVM-Wasm-Implementierung. Sie ist 5 kB groß. Die native C-Anwendung (13,3 kB) ist 8,3 kB größer als die Clang-LLVM-WebAssembly-Implementierung. Viel größer als die beiden anderen Web-Implementierungen ist die Emscripten-WebAssembly-Implementierung mit 27,4 kB.

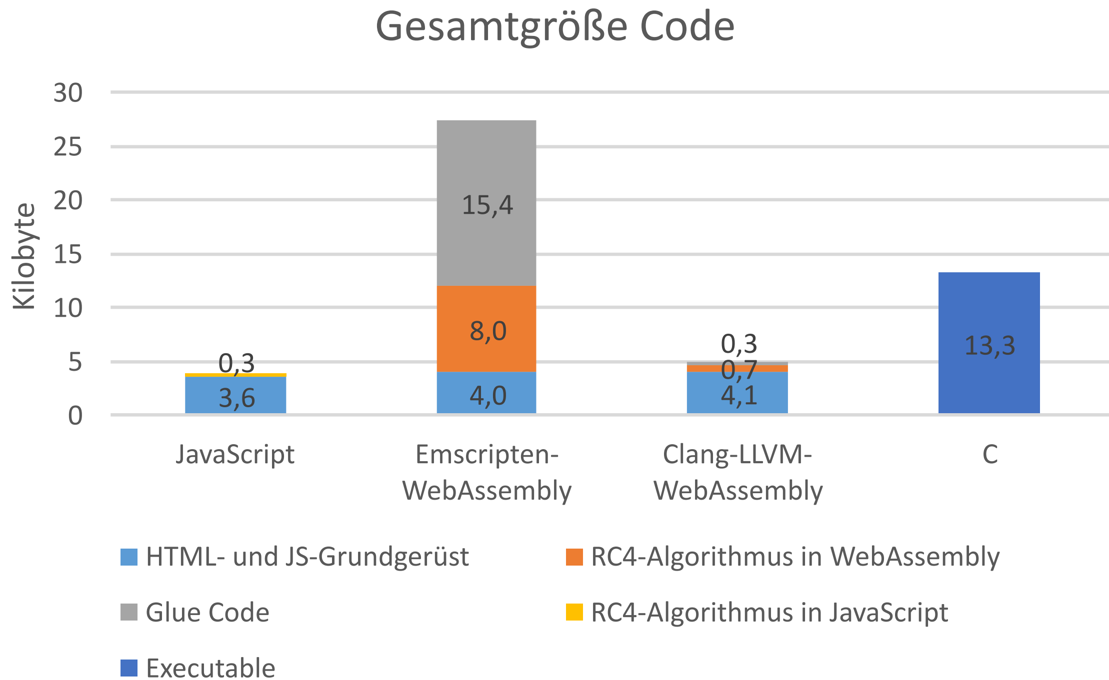


Abbildung 19: Vergleich der Gesamtgrößen der einzelnen Implementierungen

Die Gesamtgrößen der Implementierungen setzen sich aus mehreren Bereichen (Grundgerüst, Algorithmus und Glue Code) zusammen. Die Bereiche sind jeweils in eigene Dateien ausgelagert. So kann deren Größe besser erfasst werden.

Alle drei Web-Implementierungen besitzen ein Grundgerüst aus HTML- und JS-Code. Der HTML-Code enthält den sichtbaren Teil der Webseite. Der JS-Code besitzt die grundlegende Logik für die Funktionalität der Seite. Die Logik verarbeitet die Eingaben und Ausgaben. Die Verarbeitung beschränkt sich auf das Auslesen der Eingaben, das Speichern der Werte für die Verarbeitung, die Ausgabe des Ergebnisses und das Messen der benötigten Zeit. Zur Logik zählt auch das Generieren eines zufälligen Klartextes und das Reservieren des dafür benötigten Speichers. Abbildung 19 zeigt, dass die Größe der Grundgerüste ähnlich sind. Das ist darauf zurückzuführen, dass sich die Grundgerüste nur minimal unterscheiden. Für die pure JS-Implementierung ist das Grundgerüst etwas kleiner, da die Speicherreservierung automatisch stattfindet. Bei den beiden WebAssembly-Implementierungen muss der Speicher manuell reserviert werden, wodurch deren Grundgerüst etwas größer ist.

Die beiden WebAssembly-Implementierungen besitzen neben dem Grundgerüst den Glue Code, der aus JS-Code besteht. Hier zeigt sich der in der theoretischen Untersuchung beschriebene Unterschied zwischen individuell geschriebenem Glue Code in der Clang-LLVM-Toolchain und automatisch erstelltem Glue Code in der Emscripten-Toolchain.

Der individuelle Glue Code ist 0,3 kB groß, der Emscripten Glue Code 15,4 kB. Das Mehr an Größe führt neben längeren Downloadzeiten zu längeren Parsingzeiten. Beides reduziert die Ladegeschwindigkeit einer Seite. Die Website mit Clang-LLVM-WebAssembly wird daher schneller laden. Das Verhältnis zwischen Glue Code und restlichem Code in der Emscripten-Toolchain ist bei kleinen Projekten mit wenig Code schlecht. Bei großen Projekten mit viel Code wird das Verhältnis besser, da der Glue Code nicht mit dem restlichen Code mitwächst. Er bleibt, je nach verwendeten Features im Wasm-Code, gleich oder wächst nur um die verwendeten Features an.

Die Implementierung des RC4-Algorithmus ist in allen drei Web-Implementierungen in separate Dateien ausgelagert. In der puren JS-Implementierung besteht die Datei aus JS-Code, in den beiden Wasm-Implementierungen aus WebAssembly-Code. In Abbildung 19 sind die Größen der Dateien als „RC4-Algorithmus in JavaScript" und „RC4-Algorithmus in WebAssembly" dargestellt. Der JS-Code ist 0,3 kB, die Wasm-Codes 0,7 kB (Clang-LLVM-Toolchain) bzw. 8,0 kB (Emscripten-Toolchain) groß.

Aus der theoretischen Untersuchung ergab sich die Erwartung, dass Wasm-Code 20-30% kleiner ist als vergleichbarer JS-Code. Die Erwartung konnte in diesem Beispiel jedoch nicht nachvollzogen werden. Der Wasm-Code (0,7 kB) aus der Clang-LLVM-Toolchain ist mehr als doppelt so groß wie der JS-Code (0,3 kB). Der durch die Emscripten-Toolchain generierte WebAssembly-Code ist aufgrund der Menge an zusätzlichen, ungenutzten Features für einen Vergleich ungeeignet. Die Größen der WebAssembly-Codes wachsen mit steigender Menge an C-Code ähnlich an. Der mit Emscripten generierte Code ist jedoch von Beginn an größer.

Der Unterschied zwischen der theoretischen Erwartung an die Größe und der gemessenen Größe kann zum Teil auf die geringe Menge an Code der RC4-Implementierungen zurückgeführt werden. Der JS-Code ist mit 25 Zeilen in etwa so lang wie der zu Wasm kompilierte C-Code (26 Zeilen). Die Wasm-Dateien besitzen jedoch Overhead, den die JS-Datei nicht besitzt. Bei großen Dateien wird der Overhead verhältnismäßig kleiner. Der Overhead führt bei den verwendeten, kleinen Dateien dazu, dass die Wasm-Dateien größer sind als die JS-Datei. Bei mehr Code ist zu erwarten, dass die 20-30% Größenreduktion aus der theoretischen Untersuchung, erkennbar werden.

Die hier gezeigten Ergebnisse können anhand der Dateien im Anhang selbst nachvollzogen werden. Die Dateien für das HTML- und JS-Grundgerüst heißen *Grundgeruest.html*. Die Dateien für den Glue Code heißen *GlueCode.js*, und *RC4C_ohne_include.js*. Die Datei für den Algorithmus heißt *RC4.js* bei der JS-Implementierung. Bei der Wasm-Implementierung heißt sie *RC4C_ohne_include.wasm*. Die den Wasm-Dateien zu Grunde liegenden C-Dateien heißen jeweils *RC4C_ohne_include.c*. Sie sind bei beiden Wasm-Implementierungen identisch. Die HTML- und JS-Dateien sind jeweils in zwei Ausführungen vorhanden: eine nicht-minimierte Version, an der schnell Änderungen gemacht werden können, und eine minimierte Version, die für den Vergleich der Dateigrößen ge-

nutzt wird. Die minimierte Version ist immer durch den Zusatz *_minified* am Ende des Dateinamens erkennbar.

Nachdem bisher die Größen der verwendeten Dateien betrachtet wurden, wird nun die gemessene Ladegeschwindigkeit der einzelnen Web-Implementierungen betrachtet. Zum einen zeigt sich dabei, welchen Einfluss die unterschiedlichen Dateigrößen auf die Ladegeschwindigkeiten haben, und zum anderen wird ersichtlich, ob die Verwendung von WebAssembly die Ladegeschwindigkeit einer Seite erhöht.

Die Ladegeschwindigkeit wird über die Ladezeit einer Seite gemessen. Je geringer die Ladezeit ist, desto höher ist die Ladegeschwindigkeit. Die Ladezeit wird in dieser Arbeit so definiert, dass es die Zeit ist, die ab dem Aufruf der Seite aus benötigt wird, bis Inhalte ver-/ entschlüsselt werden können. Die Zeit, die zum Ver-/Entschlüsseln benötigt wird, wird als Ausführungszeit definiert. Sie wird Kapitel 3.2.4 gemessen und interpretiert.

Inhalte können bei der JS-Implementierung ver-/ entschlüsselt werden, sobald das Grundgerüst (*Grundgeruest.html*) und der Algorithmus (*RC4.js*) heruntergeladen und geparst wurden. Bei den Wasm-Implementierungen endet die Ladezeit in dem Moment, in dem das WebAssembly-Modul im JavaScript-Code zur Verfügung steht. Dafür müssen das Grundgerüst, der Glue Code (*GlueCode.js*/ *RC4C_ohne_include.js*) und die .wasm-Dateien (*RC4C_ ohne_include.wasm*) geladen und geparst bzw. kompiliert wurde. Im Anschluss daran muss das Wasm-Modul instantiiert werden. Nach der Instantiierung des Moduls endet die Ladezeit.

Abbildung 20 zeigt die Ladezeiten der JS-, Emscripten und Clang-LLVM-Implementierungen. Es ist zu erkennen, dass der Download der einzelnen Dateien den Großteil der Zeit beansprucht. Unter Download wird nicht nur die reine Zeit zur Datenübertragung zwischen Server und Client erfasst, sondern auch die Wartezeit auf den Server, nachdem der Client die Datei angefragt hat. Die Wartezeit ist die Zeit, die verstreicht, bis das erste Byte einer Datei vom Server empfangen wurde (engl. Time To First Byte (TTFB)). Sie beträgt ca. 99% der Downloadzeit. Das ist bei den verwendeten kleinen Dateien nachvollziehbar. Die reine Zeit zur Datenübertragung bei einer 100.000 kBit/s Internetverbindung und einer 4 kB Datei beträgt rechnerisch nur 0,32 ms. Die Verbindung zu `grthor.github.io` benötigt dagegen 15 ms. Dazu kommt die Zeit, die der Server benötigt die Datei bereitzustellen. Auf dem Weg vom Server zum Client werden wiederum 15 ms gebraucht. Die benötigte Zeit zum Parsen des JS-Codes ist mit 3 bzw. 2 ms vergleichsweise gering. Auch die Zeit zum Dekodieren, Kompilieren und Instantiieren ist mit 1 ms gering.

Die Ladezeit der JavaScript-Implementierung ist im Vergleich zu den Wasm-Implementierungen ca. 100 ms schneller. Der Geschwindigkeitsvorteil ist darauf zurückzuführen, dass nur zwei Dateien (*Grundgeruest.html* und *RC4.js*) vom Server heruntergeladen werden müssen. Bei den Wasm-Implementierungen müssen drei Dateien (*Grundgeruest.*

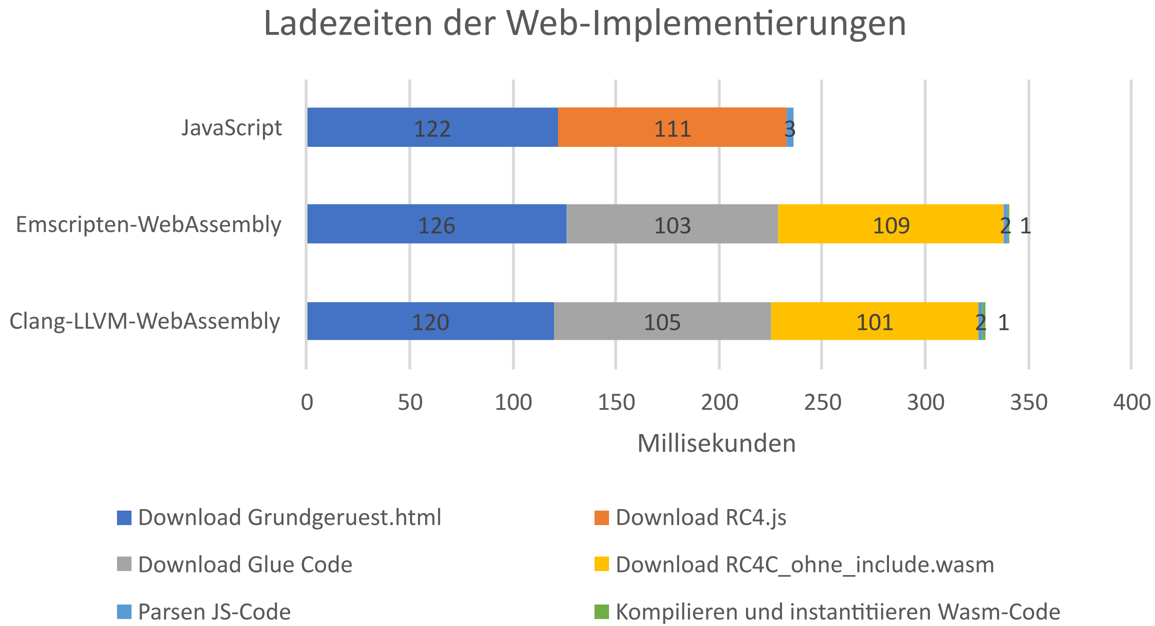


Abbildung 20: Vergleich der Ladezeiten der Web-Implementierungen

html, *GlueCode.js*/ *RC4C_ohne_include.js* und *RC4C_ohne_include.wasm*) heruntergeladen werden.

Das Parsen des JS-Codes in der JS-Implementierung benötigt ca. 3 ms. Das Parsen des Glue Codes und das Dekodieren, Kompilieren und Instantiieren des Wasm-Codes benötigt zusammen auch 3 ms. Bei der JS-Implementierung werden 0,3 kB JS-Code geparst. Bei der Clang-LLVM-Implementierung werden auch 0,3 kB JS-Code (Glue Code) geparst. Zusätzlich werden 0,7 kB Wasm-Code dekodiert und kompiliert. Bei der Emscripten-Implementierung werden 15,4 kB Glue Code geparst und 8 kB Wasm-Code dekodiert und kompiliert. Dazu kommt bei den Wasm-Implementierungen noch die Zeit, die zum Instantiieren des Wasm-Moduls benötigt wird. Bei allen drei Implementierungen ist die benötige Zeit mit 3 ms jedoch gleich. Zumindest bei den Wasm-Implementierungen müsste die Zeit verschieden sein. Es kann daher nicht gesagt werden, ob das Parsen von JS-Code schneller ist als das Dekodieren und Kompilieren von Wasm-Code. Die geringen bis nicht messbaren Unterschiede sind hauptsächlich auf die geringen Dateigrößen zurückzuführen. Sie erschweren die Erfassung der Ladezeiten. Die benötigte Zeit zur Verarbeitung der Dateien befindet sich im Bereich von Milli- und Mikrosekunden. Erschwerend kommt hinzu, dass sie nicht an einem Stück, sondern aufgeteilt stattfindet. Es ist daher schwer, genaue Daten zu erfassen, aus denen ein Rückschluss auf die Geschwindigkeit beim Parsen/Dekodieren und Kompilieren getroffen werden kann.

Die Ergebnisse lassen, trotz der geringen Unterschiede zwischen den Zeiten zum Parsen/-Dekodieren und Kompilieren von JS- bzw. Wasm-Code, einen Rückschluss zu: Für kleine

Projekte kann die Verwendung von WebAssembly, aufgrund der zusätzlich benötigten Datei (Glue Code), die Ladezeit verlängern.

Die Annahme aus der theoretischen Untersuchung zur Ladezeit von WebAssembly-Code war, dass er schneller zu Laden ist als vergleichbarer JS-Code. Um die Annahme trotz der nicht verwendbaren Testergebnisse überprüfen zu können, wird ein bereits erstellter Benchmark aus dem Internet verwendet. Bei der Auswahl des Benchmarks wurde darauf geachtet, dass er an einem realitätsnahen Beispiel durchgeführt wurde.

Der verwendete Benchmark stammt von Unity und lädt ein Projekt in WebGL [Uni18]. Das Projekt wurde im September 2018 erstellt und ist unter folgendem Link zu finden: https://blogs.unity3d.com/2018/09/17/webassembly-load-times-and-performance/. Im Benchmark werden die Ladezeiten des zu asm.js kompilierten Benchmark-Projekts und dem zu WebAssembly kompilierten Benchmark-Projekt gemessen und verglichen. Es wird optimierter JS-Code (asm.js-Code) in der der JS-Implementierung genutzt. Für die Ergebnisse bedeutet das, dass die Ladezeiten bei „normalem“, nicht optimierten JS-Code länger sind. Für die Kompilierung zu asm.js und WebAssembly wird Emscripten genutzt. Bei der Kompilierung zu Wasm wird zusätzlich Binaryen eingesetzt. Im Vergleich zu dem bisher verwendeten Beispiel befinden sich die Codemengen nicht im Bereich weniger kB, sondern im Bereich mehreren MB. Es werden 4,6 MB WebAssembly-Code und 6,1 MB asm.js-Code geladen. Das Projekt ist damit deutlich größer als das RC4-Beispiel.

Abbildung 21 zeigt die Ladezeiten des Benchmark-Projekts im Browser. Die Ladezeit endet in dem Moment, in dem die Benchmark gestartet werden kann. Die Ladezeiten wurden auf demselben Windows System mit derselben Firefox Version erfasst, wie die Ladezeiten des RC4-Beispiels. Die Vergleichbarkeit zu den Zeiten des RC4-Beispiels ist daher gegeben.

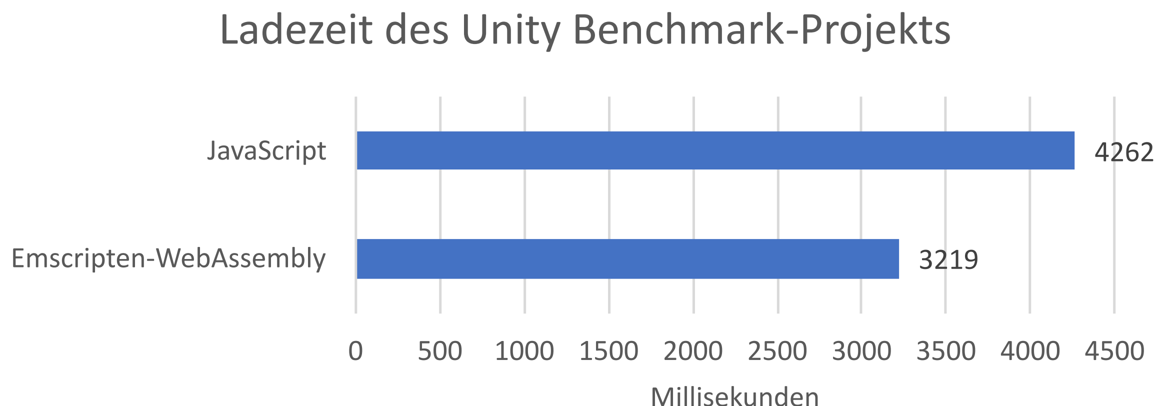


Abbildung 21: Vergleich der Ladezeiten des Unity Benchmark-Projekts

Das Beispiel von Unity umfasst 16 Dateien, die geladen, geparst/ dekodiert und kompiliert werden. Häufig finden diese Aufgaben parallel statt. Die Messung, welche Dateien wie lange geparst/dekodiert und kompiliert werden, ist daher schwer möglich. Die zur Instantiierung des Wasm-Moduls benötigte Zeit kann auch nur schwer erfasst werden, da sie verteilt stattfindet. Daher wird die Ladezeit in Abbildung 21 nicht aufgesplittet in deren Teilaufgaben dargestellt.

Abbildung 21 zeigt, dass die Ladezeit der WebAssembly-Implementierung mit 3219 ms ca. 25% geringer ist, als die der JavaScript-Implementierung mit 4262 ms. Das zeigt, dass die Verwendung von WebAssembly bei größeren Codemengen Geschwindigkeitsvorteile mit sich bringt. Die bei WebAssembly getroffenen Designentscheidungen und performancesteigernden Features gleichen die negativen Effekte des zusätzlichen Glue Codes beim Laden einer Seite mehr als aus.

Zusammenfassend sind die Dateigrößen und die Ladezeiten von Wasm bei großen Projekten geringer als bei JS. Bei kleinen Projekten sind die Dateigrößen und Ladezeiten etwas größer.

3.2.4 Ausführungsgeschwindigkeiten

Nach dem Laden einer Website beeinflusst die Ausführungsgeschwindigkeit die Usability einer Website stark. Je höher die Geschwindigkeit ist, desto besser ist die Usability. Es ist daher vorteilhaft, hohe Ausführungsgeschwindigkeiten zu erreichen. Aus der theoretischen Untersuchung kommt die Erwartung, dass die Ausführungsgeschwindigkeit einer Seite mit WebAssembly höher ist, als die einer Seite mit JavaScript.

Die theoretische Untersuchung begründete die Erwartung an eine höhere Geschwindigkeit mit mehreren Argumenten. Sie werden in diesem Absatz zusammengefasst dargestellt: Der komplette Wasm-Code wird im Vorhinein durch einen oder mehrere Compiler in der Toolchain optimiert. So kann die JSVM direkt optimierten Code ausführen. JS-Code wird hingegen erst zur Laufzeit kompiliert und nur bei häufiger Verwendung optimiert. Beide Schritte finden im selben Thread statt, in dem auch der Code ausgeführt wird. Die JIT-Kompilierung und -Optimierung reduziert durch die serielle Ausführung der Schritte die Ausführungsgeschwindigkeit des Codes. Im ungünstigsten Fall treten Fehler bei der Optimierung aufgrund der Typunsicherheit von JS auf. Tritt dieser Fall ein, muss bereits optimierter Code reoptimiert werden. Die Reoptimierung findet ebenfalls im gleichen Thread statt und verringert so die Ausführungsgeschwindigkeit weiter. Ein weiteres Argument für eine höhere Ausführungsgeschwindigkeit von Wasm-Code ist der Wegfall der Garbage Collection. Wasm benötigt keinen Garbage Collector, da der Speicher manuell gemanaged wird. Bei JS hingegen unterbricht der Garbage Collector die Ausführung des Codes, um Speicher freizugeben. Die Garbage Collection arbeitet ebenfalls im selben Thread und reduziert dadurch auch die Ausführungsgeschwindigkeit

von JS-Code. Aus diesen Gründen kommt die theoretische Untersuchung zu der Erwartung, dass WebAssembly-Code schneller ausgeführt wird als JavaScript-Code. Diese Erwartung wird im Folgenden überprüft.

Die Ausführungsgeschwindigkeiten werden anhand der RC4-Implementierungen gemessen. Die Implementierungen setzen sich aus mehreren Programmphasen zusammen. Zuerst wird die Eingabe aus der GUI (bei den Web-Implementierungen) bzw. der Konsole (bei der C-Implementierung) eingelesen und in ein intern verwendbares Format konvertiert. Daraufhin wird ein zufälliger Klartext in der eingegebenen Länge erzeugt. Der Klartext wird darauf folgend mit der RC4-Stromchiffre verschlüsselt. Abschließen werden die ersten 16 Byte des Geheimtextes in der GUI ausgegeben.

Bei der Messung der Ausführungsgeschwindigkeit wird nur die Zeit zur Verschlüsselung des Klartextes erfasst. So sind die Messwerte unverfälscht. Würde die komplette Ausführung des Programms gemessen, würden etwaige Geschwindigkeitsvor-/ -nachteile durch die Zeiten der anderen Programmphasen verfälscht.

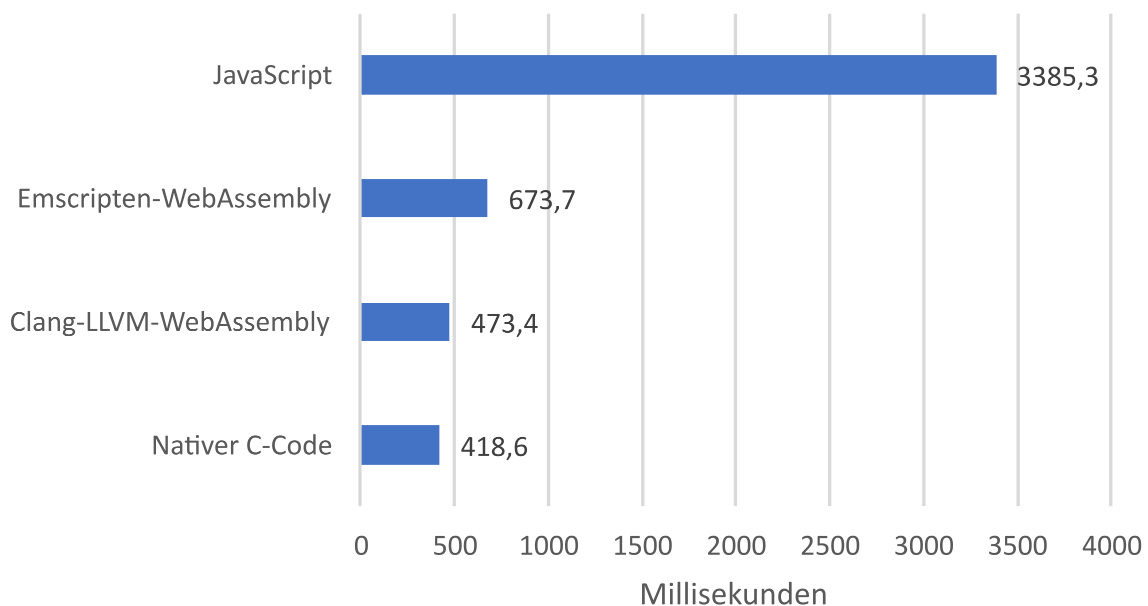


Abbildung 22: Vergleich der Laufzeiten

Abbildung 22 zeigt die Zeiten der einzelnen Implementierungen, die zur Verschlüsselung von 128 MB an zufälligen Daten benötigt werden. Es ist zu erkennen, dass JavaScript mit 3385,3 ms dafür deutlich am längsten benötigt. Darauf folgt die Emscripten-WebAssembly Implementierung mit 673,7 ms. Sie ist ca. 5 mal schneller als die JS-

Implementierung. Am zweitschnellsten ist die Clang-LLVM-WebAssembly-Implementierung. Sie benötigt 473,4 ms und ist damit rund 7 mal schneller als die Emscripten-WebAssembly-Implementierung. Am schnellsten ist die native C-Implementierung. Sie verschlüsselt die Daten in 418,6 ms.

Die gemessenen Zeiten zur Verschlüsselung bestätigen die Erwartungen aus der theoretischen Untersuchung. WebAssembly ist deutlich schneller als JavaScript. Es benötigt in diesem Beispiel höchstens ein Fünftel der Zeit von JS. Insbesondere die Clang-LLVM-Implementierung ist mit nur 54,8 ms mehr Laufzeit nur 13% langsamer als nativer C-Code. Damit ist WebAssembly annähernd so schnell wie herkömmlicher C-Code.

3.3 Fazit

Die theoretische Untersuchung hat viele Details über die Architektur von WebAssembly dargelegt. Dadurch wurden Geschwindigkeitsvor- und -nachteile von Wasm deutlich. Darauf aufbauend konnten Erwartungen an die Geschwindigkeit formuliert werden. In der praktischen Untersuchung wurden die Erwartungen anhand von realen Beispielen überprüft. Die Implikationen, die sich aus der Überprüfung für den realen Einsatz von WebAssembly ergeben, werden im Folgenden dargestellt.

Der Einfluss von WebAssembly auf die Ladegeschwindigkeit einer Website hängt von der Projektgröße ab. Bei kleinen Projekten kann die Verwendung von WebAssembly dazu führen, dass eine Website langsamer lädt. Das Wasm-RC4-Beispiel, als kleines Projekt, benötigt ca. 100 ms mehr als eine vergleichbare JS-Implementierung. Bei mittleren und großen Projekten führt die Verwendung von Wasm zu einer höheren Ladegeschwindigkeit. Das Unity-Benchmark-Projekt, als großes WebAssembly-Projekt, lädt ca. 1000 ms schneller als dessen pure JavaScript-Variante.

Die Ausführungsgeschwindigkeit einer Website profitiert eindeutig von der Verwendung von WebAssembly. WebAssembly spielt hier den Vorteil aus, dass der Code in der Toolchain optimiert und beim Laden vollständig in Maschinencode kompiliert wird. Soll Code ausgeführt werden, so steht er sofort in optimierter Form bereit. Die Ausführungsgeschwindigkeit des RC4-Beispiels aus der Clang-LLVM-Toolchain ist ca. 5 mal so hoch wie die der puren JavaScript-Implementierung. Sie ist nur 13% langsamer als eine native C-Anwendung.

Zusammenfassend kann aus Sicht der Geschwindigkeit eine klare Empfehlung zur Verwendung von WebAssembly gegeben werden. Die Teils längeren Ladezeiten bei kleinen Projekten beeinflussen die Usability einer Seite nur gering. Dieser leicht negative Einfluss wird durch die späteren Usability-Gewinne aufgrund einer höheren Ausführungsgeschwindigkeit mehr als ausgeglichen.

Für die Zukunft ist zu erwarten, dass die Geschwindigkeit von WebAssembly weiter steigen wird. Es sind bereits diverse lade- und ausführungsgeschwindigkeitssteigernde Features, wie das Cachen von kompiliertem Wasm-Code oder SIMD, geplant.

4 Evaluierung der Sicherheit von WebAssembly

Die Sicherheit einer Technologie ist ein entscheidendes Kriterium für oder gegen deren Nutzung. Insbesondere bei Webtechnologien hat das Kriterium „Sicherheit“ eine hohe Bedeutung. Die Vertrauenswürdigkeit vieler Seiten im Internet kann nicht überprüft werden. Daher ist es wichtig, dass der Besuch unbekannter Webseiten sicher ist. Damit das möglich ist, müssen die genutzten Technologien sicher sein.

WebAssembly wurde mit dem Ziel designt, fehler- oder schadhaften Code sicher auf dem System eines Nutzers ausführen zu können [W3C18a]. Dabei profitiert Wasm einerseits von bereits vorhandenen Sicherheitsmechanismen der Browser [Ber18b], anderseits bringt es Sicherheitsprobleme, die bisher untypisch für Webtechnologien waren, ins Web [Ber18a].

Im Folgenden wird die Sicherheit von WebAssembly analysiert. Die Analyse beginnt mit allgemeinen Informationen zur Sicherheit von WebAssembly. Im darauffolgenden Kapitel wird speziell auf Sicherheitsprobleme eingegangen, die durch das manuelle Speichermanagement in WebAssembly entstehen. Daraufhin wird die Control Flow Integrity genauer betrachtet. Der Control Flow ist durch das manuelle Speichermanagement manipulierbar. In Kapitel 4.4 wird ein umstrittenes Anwendungsfeld von WebAssembly, das Crypto-Mining, beleuchtet. Dabei wird darauf eingegangen, was Crypto Mining ist, welche Chancen und Risiken es birgt und ob es rentabel ist. Daraufhin wird ein klar böswilliges Anwendungsgebiet betrachtet, die Obfuskation von Schadcode mithilfe von WebAssembly. Abschließend wird in Kapitel 4.6 eine Empfehlung gegeben, ob WebAssembly aus sicherheitstechnischen Gesichtspunkten verwendet werden sollte oder nicht.

4.1 Allgemeine Informationen zur Sicherheit

Wasm-Code wird in einer Sandbox ausgeführt. Die Sandbox ist dieselbe, wie die für JS-Code. Sie kann nur über entsprechende APIs verlassen werden [W3C18a]. Die Nutzung einer Sandbox kapselt den Wasm-Code von der Browserumgebung ab. WebAssembly selbst kann so nicht mit seiner Umgebung interagieren. Für Interaktionen benötigt es Glue Code, der auch in der Sandbox ausgeführt wird und dadurch auch von seiner Umgebung abgekapselt ist. Bezogen auf das Verlassen der Sandbox ist WebAssembly daher so sicher wie JavaScript [Ber18b].

Unabhängig vom Einsatzort von WebAssembly, sei es in einer Webumgebung (bspw. einem Webbrowser) oder einer Nicht-Webumgebung (bspw. Wasmtime), unterliegt WebAssembly den Sicherheitsrichtlinien der Umgebung. In einer Webumgebung kann das

die Same Origin Policy sein, in einer Nicht-Webumgebung das POSIX-Sicherheitsmodell [W3C18a].

WebAssembly ermöglicht durch seine Leistungsfähigkeit die Verlagerung von Anwendungen weg von einem Client-Server- oder cloudbasierten Modell hin zu einem nur clientbasierten Modell. Dadurch können sicherheitskritische Aufgaben, wie die Generierung von RSA-Schlüsseln rein lokal ausgeführt werden. Die Verlagerung hin zum Client erhöht die Privatsphäre, da kein möglicherweise kompromittierter Server mehr eingebunden ist. Durch die Verlagerung hin zum Client wird auch die Kommunikation zwischen Client und Server reduziert, wodurch die Angriffsfläche der Anwendung minimiert wird.

Jede weitere Technologie in einer Webanwendung erhöht jedoch auch die Angriffsfläche der Anwendung. Das ist auch beim Einsatz von WebAssembly der Fall. Bei einer Entscheidung, ob WebAssembly eingesetzt werden soll oder nicht, sollte dieser Punkt beachtet werden [Ber18b].

4.2 Speichersicherheit

WebAssembly bringt Sprachen mit manuellem Speichermanagement ins Web. Bei manuellem Speichermanagement werden, im Vergleich zu automatischem Speichermanagement, Speicheradressen vom Programmierer einer Anwendung verwaltet. Das hat Vor- und Nachteile. Ein Nachteil ist, dass es diverse Sicherheitsrisiken mit sich bringt. Die Risiken sind bereits seit den neunziger Jahren bekannt. Bisher waren sie jedoch untypisch für das Webumfeld. Das schürt Unsicherheiten über die Sicherheit von WebAssembly [Ber18b]. Daher wird das manuelle Speichermanagement und seine Auswirkung auf die Sicherheit von Webanwendungen in diesem Kapitel genauer betrachtet.

WebAssembly nutzt einen linearen Speicher. Ein linearer Speicher ist ein langer, zusammenhängender Speicherbereich, ähnlich wie ein Array. Er wird im Glue Code angelegt und ist als JS ArrayBuffer realisiert. Auf ihn können der WebAssembly- und JavaScript-Code zugreifen. Das manuelle Speichermanagement bezieht sich nur auf den linearen Speicher. Das heißt, alle Sicherheitsrisiken im Zusammenhang mit dem Speichermanagement beziehen sich nur auf den linearen Speicher. Anderer Speicher kann nicht beschrieben oder ausgelesen werden [Ber18a].

Eine Schwachstelle von manuellem Speichermanagement sind Buffer Overflows. Sie sind untypisch für herkömmliche Webanwendungen. Wenn sie im Web auftreten, finden sie sich im Backend einer Anwendung auf einem Server. Durch WebAssembly können sie jetzt auch im Frontend auftreten [Ber18a; LKP20].

Der in Listing 1 gezeigte Code funktioniert korrekt für Eingaben, die gleich oder kürzer sind als 10 Zeichen. Die Funktion liefert nur 0 zurück, wenn `input = "0123456789`

`"` ist. Ist die Eingabe jedoch länger als 10 Zeichen, so wird die Variable `text[10]` überschrieben. Das liegt daran, dass `buf[10]` nur 10 Zeichen aufnehmen kann. Alle weiteren Zeichen werden in Variablen geschrieben, die im Stack folgen. Bekommt die Funktion die Eingabe `"11111111111111111111"` gibt sie fälschlicherweise 0 zurück. `"0123456789"` und `"11111111111111111111"` stimmen zwar nicht überein, jedoch überschreibt `sprintf(buf, input)` die Variable `text` mit `"1111111111"`. Hier tritt ein Buffer Overflow auf. In der Zeile `return strcmp(text, buf)` haben `text` und `buf` den Wert `"1111111111"`. Die Funktion liefert deshalb 0 zurück.

```
/* Die Funktion prueft, ob der eingegebene String gleich dem
   gespeicherten String ist. */
int check (char* input) {
        char text[10] = "0123456789";
        char buf[10];
        /* sprintf kopiert input in buf */
        sprintf(buf, input);
        /* Gibt 0 bei gleichen Strings zurueck. */
        return strcmp(text, buf);
}
```

Listing 1: C-Code mit Buffer-Overflow-Schwachstelle

Mit Hilfe von Buffer Overflows können alle, bis zum Auftreten des Overflows, im Stack gespeicherten Daten überschrieben werden. Ähnlich des in Listing 1 gezeigten Beispiels ist die Manipulation von SQL-Abfragen oder das Überschreiben von Funktionspointern möglich [Ber18a].

Herauszufinden, wo welche Variable im Speicher von WebAssembly liegt, ist einfach möglich. Die ausgeführte .wasm-Datei kann dekompiliert und die Speicheradressen ausgerechnet werden. Da auf Verfahren wie Address Space Layout Randomization zur zufälligen Verteilung von Variablen im Speicher verzichtet wurde, liegen die Variablen immer an denselben Stellen [Mus+19b].

Das Verändern von Funktionspointern beeinflusst den Ablauf eines Programms. Dadurch kann es zu unerwartetem und schadhaftem Verhalten kommen. Das Verändern des Programmablaufs wird als Control Flow Manipulation bezeichnet. Die Ausgangssituation für Control Flow Manipulation ist, dass der Control Flow erst zur Laufzeit bestimmt wird. Das ist der Fall, wenn Funktionen über ihre Speicheradresse aufgerufen werden. Die Speicheradressen können dadurch erst zur Laufzeit bestimmt werden [W3C18a; LKP20]. Listing 2 zeigt Code, in dem durch einen Buffer Overflow ein Funktionspointer überschrieben werden kann.

Wird die Funktion `function_pointer_overwrite(char* str)` mit einem mehr als 50 Zeichen langen String aufgerufen, so findet ein Buffer Overflow bei `sprintf(...)` statt. Dadurch wird der mit `function_pointer = loginUser` im Stack gespeicherte

Funktionspointer überschrieben. Im Vorfeld des Angriffs muss die Adresse der Funktion `loginAdmin()` herausgefunden werden. Der Funktion `function_pointer_overwrite` kann dann als 51. Char die Adresse übergeben werden. Sie überschreibt dann den Funktionspointer. Der Control Flow des Programms ist damit verändert.

```
void loginUser () {
        /* Nutzerspezifischer Code */
}

void loginAdmin () {
        /* Adminspezifischer Code */
}

void function_pointer_overwrite (char* str) {
        void* function_pointer;
        char buf[50];
        function_pointer = loginUser;
        sprintf(buf, "Es wurde folgender Text eingegeben: %s",
            str);
        ((void)(*)(void) function_pointer)();
}
```

Listing 2: C-Code mit Schwachstelle zum Überschreiben eines Funktionspointers

Mehrere Mechanismen verhindern in WebAssembly, dass der Control Flow wahllos beeinflusst werden kann. Die Mechanismen werden im folgenden Kapitel beschrieben.

Neben Funktionspointern können auch Pointer auf Daten überschrieben werden. Sie können dadurch auf nicht initialisierte oder freigegebene Speicherbereiche zeigen. Das kann zu unerwartetem Verhalten und im schlimmsten Fall zu einem Absturz des Programms führen. Pointer auf Bereiche außerhalb des Speichers werden über Bound Checks blockiert und führen zu einem Absturz des Programms [W3C18a].

Das unbemerkte Überschreiben von Daten im Stack kann durch den Einsatz von Stack Canaries erschwert werden. WebAssembly verzichtet jedoch auf den Einsatz dieser. Stack Canaries sind zufällige Werte, die zwischen die Daten im Stack gelegt werden. Tritt ein Buffer Overflow auf, so wird mindestens ein Stack Canarie überschrieben. Bemerkt die Anwendung, dass ein Stack Canarie überschrieben wurde, kann es Gegenmaßnahmen einleiten. Auf diese Weise erschweren Stack Canaries die Ausnutzung von Buffer-Overflow-Schwachstellen. Wie bereits erwähnt, werden sie in WebAssembly jedoch nicht eingesetzt [Mus+19b].

Return Pointer können in WebAssembly nicht überschrieben werden. Der Stack mit den Rücksprungadressen (engl. Call stack) wird in einem gesonderten, nicht beschreibbaren Bereich gespeichert. Es ist daher nicht möglich, einen Buffer Overflow im Call Stack zu

erzeugen. Stack Smashing Protection ist dadurch nicht nötig. Dass der Call Stack nicht beschreibbar ist, vermeidet zudem Return-oriented Programming [W3C18a].

4.3 Control Flow Integrity

Control Flow Integrity (CFI) bezeichnet Mechanismen, den Programmablauf gegen unerwünschte Änderungen abzusichern [Wik21]. Sie sind ein elementarer Bestandteil des Sicherheitskonzepts von WebAssembly. In Kapitel 4.2 wird an einem Beispiel gezeigt, wie der Control Flow manipuliert und so die CFI verletzt werden kann. In diesem Kapitel werden die in WebAssembly vorhandenen Mechanismen vorgestellt, die die Manipulation des Control Flows erschweren. Es wird darauf eingegangen, wie die Mechanismen funktionieren und wie der Schutz gegen Control Flow Manipulation in einer Anwendung durch den Anwendungsentwickler verbessert werden kann.

In WebAssembly gibt es drei Arten von Funktionsaufrufen: Direkte Funktionsaufrufe, indirekte Funktionsaufrufe und Returns. Direkte Funktionsaufrufe sind Aufrufe zu Funktionen, bei denen die Adresse der aufgerufenen Funktion bereits bei der Kompilierung ermittelt werden kann. Die Adresse steht die komplette Laufzeit über fest. Ein Beispiel für einen direkten Funktionsaufruf ist `loginUser()` in Listing 2. Indirekte Funktionsaufrufe sind Aufrufe zu Funktionen, bei denen die Adresse der aufgerufenen Funktion erst zur Laufzeit ermittelt werden kann. Ein Beispiel für einen indirekten Funktionsaufruf ist `((void)(*)(void)function_pointer)()`. Kontextinformationen über den Aufruf können Listing 2 entnommen werden [Chr19]. Returns sind Aufrufe zurück in den Call Stack. Sie finden statt, nachdem eine Funktion durchlaufen wurde und zurück in die aufrufende Funktion gesprungen wird. Die drei Arten von Funktionsaufrufen sind jeweils unterschiedlich gegen Control Flow Manipulation abgesichert [W3C18a].

Direkte Funktionsaufrufe werden über einen Control Flow Graph abgesichert. Der Control Flow Graph wird bei der Kompilierung erzeugt. Er besteht aus allen möglichen Folgen von Funktionsaufrufen, die zur Laufzeit auftreten können. Zur Laufzeit wird bei einem Funktionsaufruf überprüft, ob die Aufruffolge im Control Flow Graph enthalten ist. Ist sie das, wird der Aufruf zugelassen. Ist sie nicht enthalten, beendet sich das Programm. Um den Control Flow Graph zu erzeugen ist es notwendig, die Adressen der Funktionen zu kennen. Daher können nur direkte Funktionsaufrufe darüber abgesichert werden [W3C18a].

Returns werden gegen Manipulation abgesichert, indem der Call Stack in einem nicht beschreibbaren Bereich des Speichers abgelegt ist. Die dort gespeicherten Adressen können daher nicht überschrieben werden [W3C18a].

Indirekte Funktionsaufrufe werden über eine Funktionstabelle und Singnaturprüfungen abgesichert. Sie bieten jedoch keinen vollständigen Schutz, wie das Beispiel in Listing

2 zeigt. Im Allgemeinen führt ein indirekter Funktionsaufruf Daten an einer beliebigen Stellen im Speicher aus. Im korrekten Anwendungsfall liegt an der auszuführenden Stelle im Speicher die gewünschte Funktion. Im schlechtesten Fall liegt dort eine Funktion, die ein Angreifer zuvor eingeschleust hat. So kann Schadcode mithilfe von indirekten Funktionsaufrufen ausgeführt werden [Cla17e; Ber18a].

Das beliebige Ausführen von Daten in WebAssembly wird durch den Einsatz einer Funktionstabelle abgesichert. Die Funktionstabelle bildet eine Zwischenschicht beim Aufruf von Daten durch einen indirekten Funktionsaufruf. Der Funktionsaufruf ruft nicht mehr Daten an einer beliebigen Speicheradresse auf, sondern einen Index in der Funktionstabelle. Die Funktionstabelle enthält alle Speicheradressen der Funktionen des WebAssembly-Codes. Sie holt die Speicheradresse der Funktion anhand des Index aus der Tabelle und führt die Funktion an der geholten Stelle im Speicher aus. Die Funktionstabelle und die zugehörigen Funktionen liegen außerhalb des beschreibbaren Speichers und können daher zur Laufzeit nicht manipuliert werden. So können nur Funktionen ausgeführt werden, die bereits zur Kompilierung des Codes existierten. Das verhindert das Ausführen von eingeschleustem Code in WebAssembly [Cla17e]. Die Manipulation des aufgerufenen Index ist jedoch weiterhin möglich [Ber18a].

Das Manipulieren des Index bei einem indirekten Funktionsaufruf ist möglich, beispielsweise durch einen Buffer Overflow. Ein Angreifer kann beliebige vorhandene Funktionen ausführen. Signaturprüfungen schränken die Möglichkeit ein, beliebige Funktionen auszuführen. Für jede Funktion wird eine Signatur erstellt. Die Signatur besteht aus der Anzahl und den Typen der Funktionsargumente. Bei einem Funktionsaufruf wird überprüft, ob der Aufruf mit derselben Argumentanzahl und denselben Argumenttypen stattfindet. Die Signatur verhindert nur das Ausführen von Funktionen mit unterschiedlichen Argumenten. Aufrufe von Funktionen, die dieselbe Argumentanzahl und dieselben Argumenttypen nutzen, können trotz Signaturprüfung manipuliert werden. Gegen die Manipulation der Indexe auf Funktionen mit denselben Argumenten besitzt WebAssembly keinen Schutz [Ber18a].

Die vier beschriebenen Mechanismen gegen die Manipulation des Control Flows (Nutzung eines Control Flow Graphs, der geschützte Call Stack, die Verwendung einer Funktionstabelle und die Singnaturprüfung) bieten einen umfassenden Schutz. Sie sind standardmäßig in WebAssembly aktiviert. Bis auf eine Schwachstelle, die die Manipulation des Control Flows bei der Verwendung indirekter Funktionsaufrufe zulässt, ist WebAssembly gut gegen Control Flow Manipulation geschützt.

Der Schutz gegen Control Flow Manipulation kann verbessert werden, indem das Flag -$fsanitize = cfi$ beim Kompilieren gesetzt wird. Es aktiviert die Mechanismen zur Sicherung der CFI aus LLVM [Cla21]. Die Mechanismen sind feingranularer und umfassender als die in WebAssembly standardmäßig eingebauten Verfahren. Die Nutzung geht mit einer leichten Reduzierung der Ausführungsgeschwindigkeit einher. Das Flag ist

sowohl in der Emscripten-Toolchain als auch in der Clang-LLVM-Toolchain verfügbar [W3C18a; Cla21].

Features zur weiteren Absicherung der CFI in WebAssembly sind geplant. Dazu zählt ein Verfahren ähnlich wie Address Space Layout Randomization (ASLR), sowie die Einführung von Atomics und Bounded Pointers (Fat Pointers) [W3C18a].

4.4 In-Browser Crypto Mining

Eines der ersten Anwendungsgebiete für WebAssembly war das Crypto Mining. Bereits im September 2017, ein halbes Jahr nach dem Release von WebAssembly, erschien der erste WebAssembly-basierte Service zum Minen von Kryptowährungen auf Websites: Coinhive (Abbildung 23). Coinhive erlaubt die Rechenleistung der Besucher zu nutzen, um damit Kryptowährungen zu schürfen (engl. minen). Der Service wurde 2019 abgeschaltet. Aktuell gibt es jedoch diverse ähnliche Angebote wie Coinhive, beispielsweise CoinImp, Moonify, Miner.Ad [Mus+19b].

Abbildung 23: Logo des Mining-Services Coinhive

Crypto Mining bezeichnet das Durchführen aufwändiger Berechnungen, um neue Blöcke an die Blockchain einer Kryptowährung anzufügen. Die Berechnung ist rechenaufwändig und somit stromintensiv. Die Stromkosten sind der Hauptkostenfaktor beim Minen von Kryptowährungen. Der Lohn fürs Minen sind Anteile an der Währung [Rüt+18].

In-Browser Crypto Mining bezeichnet das Schürfen von Kryptowährungen im Browser. Dabei wird die Rechenleistung des Websitebesuchers für die Berechnungen genutzt. Den für die Berechnungen benötigten Strom bezahlt der Besucher, da die Berechnungen auf seinem Rechner ausgeführt werden. Der Lohn, die Anteile an der Kryptowährung, kommt jedoch nicht dem Besucher zu, sondern dem Betreiber der Website. Daraus ergibt sich für den Websitebetreiber eine Erlösquelle [Rüt+18].

Crypto Mining im Browser ist keine neue Idee. Erste Ansätze gab es bereits 2007. Das Mining fand dabei in JavaScript statt. Die Motivation für die Entwicklung des Skriptes war bereits damals die Steigerung des Erlöses durch den Traffic auf einer Website [Kre18]. Aufgrund der geringen Geschwindigkeit von JavaScript und der somit geringen Rentabilität verbreitete sich die Idee jedoch nicht weiter. Mit der Entwicklung von WebAssembly und der damit verfügbaren höheren Geschwindigkeit stieg die Rentabilität und die Idee verbreitete sich. Forscher der TU Braunschweig haben 2019 die Verbreitung von Crypto-Mining im Browser untersucht. Sie kommen zu dem Ergebnis, dass auf 0,25% der Top eine Million Websites von Alexa, also auf mindestens 2500 Websites, Crypto-Mining-Skripte laufen [Mus+19b].

In-Browser Crypto Mining bietet Chancen und Risiken. Als Chance ist die Reduzierung der Anzahl an Werbeanzeigen auf einer Website zu nennen. Ein Websitebetreiber kann die Anzahl der Anzeigen reduzieren, indem er die bisher durch Werbung erzielten Erlöse durch die Erlöse aus dem Crypto Mining ersetzt. Das Minen ist erlaubt, solange der Besucher dem zustimmt. Das Minen ohne die Zustimmung des Besuchers ist ein Risiko und wird als Cryptojacking bezeichnet. Die Rechenleistung des Endgeräts des Besuchers wird „entführt“ und zum Schürfen von Kryptowährungen genutzt. Der Besucher leidet dabei unter einer verringerten Leistung seines Endgeräts und unter einem höheren Strombedarf, den er finanzieren muss [Mus+19b].

In-Browser Crypto Mining lässt eine Hoffnung auf werbefreie Webseiten entstehen. Die Hoffnung muss jedoch gedämpft werden. Nach BitcoinMag [Bit17] und Technikbrennpunkt [Tec18] ist das Minen nicht rentabel. BitcoinMag errechnete 2017 für einen durchschnittlichen Laptop und eine Verweildauer von 3 Minuten auf einer Website einen Ertrag von 0,00000286 Monero (XMR) pro Besuch. Bei einem Kurs von 207.76 € pro Monero Coin (17.02.2021) sind das 0,0005941936 €. Hochgerechnet auf 1000 Besuche à 3 Minuten ergibt das einen Erlös von ca. 0,59 €. Daraus zeigt sich, dass In-Browser Crypto Mining nur bei langen Verweildauern, starken Endgeräten, vielen Besuchern und hohen Umrechnungskursen rentabel sein kann [Bit17].

Der Schutz vor Cryptojacking, dem unerlaubten Schürfen von Kryptowährungen auf dem eigenen Endgerät, ist eine Herausforderung. Aktuelle Gegenmaßnahmen beschränken sich auf die Nutzung von Blacklists. Die Adblocker Adblock Plus und uBlock sowie die Chrome Extensions NoCoin und MinerBlock verwenden beispielsweise Blacklists. Sie bieten jedoch keinen vollständigen Schutz gegen Cryptojacking. So blockieren Blacklists nur 45 bis 63% der Miner. Ein vollständiger Schutz gegen Cryptojacking ist das Abschalten von JavaScript und somit auch das Abschalten von WebAssembly. Das hat jedoch den Nachteil, dass alle Seiten, die JS und/oder Wasm nutzen, nicht mehr korrekt funktionieren [Mus+19b].

4.5 Obfuskation von Schadcode

Ein missbräuchliches Anwendungsgebiet von WebAssembly ist die Verschleierung (Obfuskation) von Schadcode. Bei der Obfuskation von Schadcode wird HTML- und JS-Code in den WebAssembly-Code eingebettet. Es wird ausgenutzt, dass WebAssembly neu ist und Wasm-Code nicht von allen Analysetools gescannt wird [Mus+19a].

Forscher der TU Braunschweig haben 2019 in einer Studie zu den Einsatzgebieten von WebAssembly herausgefunden, dass 0,01% der Top eine Million Websites von Alexa, also mindestens 1000 Websites, WebAssembly nutzen, um Schadcode zu verschleiern [Mus+19a].

Der verschleierte Schadcode in den Top eine Million Websites von Alexa hatte unterschiedliche Aufgaben. Bei der Analyse haben sich zwei Hauptanwendungsgebiete herauskristallisiert. Ein Teil sollte ein Pop-under öffnen. Ein Pop-under ist ein Fenster im Hintergrund. Es hat das Ziel, möglichst lange unbemerkt offen zu bleiben. Es kann bspw. zum Minen von Kryptowährungen genutzt werden. Da das Fenster lange geöffnet ist, erhöht sich der Ertrag für den Websitebetreiber. Das zweite Anwendungsgebiet ist das Umgehung von AdBlockern, um den Ertrag einer Seite durch Werbung zu erhöhen [Mus+19a].

4.6 Fazit

Die Sicherheit von WebAssembly wurde zu Beginn des Kapitels aus einer allgemeinen Sicht analysiert. Dabei hat sich gezeigt, dass Wasm, was das Verlassen der JSVM betrifft, genauso sicher ist wie JavaScript. Das ist darauf zurückzuführen, dass der Code beider Sprachen in derselben VM ausgeführt wird. WebAssembly unterliegt weiterhin der Policy des Embedders. Im Browser ist das die Same-Origin-Policy. Sie verhindert, dass WebAssembly, wie auch JS, auf andere Tabs zugreifen kann. Wasm kann nur mit Daten aus „seinem" Tab arbeiten. Aufgrund der Leistungsfähigkeit von WebAssembly kann dessen Einsatz die Sicherheit von Webanwendungen positiv beeinflussen. Anwendungen, die aufgrund leistungsintensiver Aufgaben ein Client-Server-Modell nutzen, können auf ein clientbasiertes Modell umgestellt werden. Dadurch kann die Kommunikation zwischen Client und Server reduziert und die Angriffsfläche der Anwendung minimiert werden. Auf der anderen Seite vergrößert der Einsatz jeder weiteren Technologie in einer Anwendung deren Angriffsfläche.

Die Speichersicherheit ist der Hauptpunktkritikpunkt in der Diskussion um Sicherheitsbedenken gegen WebAssembly. Durch das manuelle Speichermanagement von Wasm sind Angriffe möglich, die bisher untypisch für das Webumfeld waren. Es konnte aber festgestellt werden, dass aufgrund des Designs von WebAssembly die Angriffsfläche klein

gehalten wird. Buffer Overflows in WebAssembly sind jedoch möglich und stellen einen Schwachpunkt dar. Durch sie können Daten im Stack manipuliert werden. Durch den Verzicht auf Stack Canaries können sie unbemerkt stattfinden. Buffer Overflows können genutzt werden, um beispielsweise SQL-Injections durchzuführen oder Funktionspointer zu überschreiben. Eine Absicherung gegen Buffer Overflows, beispielsweise eine korrekte Eingabevalidierung, sollte in WebAssembly-basierten Anwendungen daher von Anwendungsentwicklern implementiert werden.

Durch Schwachstellen im Speichermanagement kann der Control Flow von WebAssembly-Anwendungen manipuliert werden. Das ist jedoch nur bei indirekten Funktionsaufrufen möglich. Direkte Funktionsaufrufe und Returns können nicht manipuliert werden. Sie werden durch die Nutzung eines Control Flow Graphs und geschützte Speicherbereiche abgesichert. Um die Manipulation von indirekten Funktionsaufrufen zu erschweren, nutzt WebAssembly Funktionstabellen und Signaturprüfungen. Die Signaturprüfung kann durch das Setzen des Flags $-fsanitize = cfi$ beim Kompilieren verbessert werden. Das Risiko der Control Flow Manipulation wird in WebAssembly gering gehalten.

Mit WebAssembly kommt das Anwendungsfeld des Crypto Mining in den Browser. In-Browser Crypto Mining stellt Chancen und Risiken dar. Zum einen können Besucher auf werbefreie Webseiten hoffen, da die Monetarisierung der Inhalte über Crypto Mining, anstelle von Werbung, stattfindet. Zum anderen stellt das ungefragte Mining ein Risiko dar, da es das Endgerät des Besuchers belastet und dessen Stromrechnung erhöht. Einen vollständigen, anwenderfreundlichen Schutz gegen ungefragtes In-Browser Crypto Mining gibt es aktuell nicht. Es kann diskutiert werden, ob Crypto Mining im Browser ein Risiko darstellt oder nicht. Da diese Frage jedoch nicht geklärt ist, fließt die Chance bzw. das Risiko des Crypto Mining mit Hilfe von WebAssembly nicht in die Bewertung, ob WebAssembly aus sicherheitstechnischen Aspekten genutzt werden sollte, ein.

Die Verschleierung von Schadcode in WebAssembly stellt aktuell eine Bedrohung dar. Da WebAssembly neu ist, unterstützen nicht alle Analysetools die Analyse von Wasm-Code. Das ermöglicht die Obfuskation von Schadcode darin. Wie sich die Bedrohung in Zukunft entwickelt, bleibt abzuwarten. Es ist ein Rennen zwischen Angreifern, die immer bessere Techniken zur Obfuskation entwickeln, und Entwicklern von Analysetools, die immer bessere Analysen von Wasm-Code ermöglichen.

Zusammenfassend kann WebAssembly aus sicherheitstechnischen Aspekten verwendet werden. Es ist, was das Verlassen der Browserumgebung betrifft, so sicher wie JavaScript. Ein Aspekt, der jedoch beachtet werden muss, ist die korrekte Eingabevalidierung durch den Anwendungsentwickler, da ansonsten Buffer Overflows möglich sind.

5 Möglichkeiten zur Oberflächengestaltung mit WebAssembly

Attraktive Websites sind wichtig, um Besucher darauf zu halten. Für Websitebetreiber ist es daher bedeutsam, ihre Website visuell ansprechend zu gestalten. Für die Gestaltung existieren bereits diverse Technologien und Frameworks. Mit dem Auftreten von WebAssembly stellt sich die Frage, ob es zur Gestaltung einer Website genutzt werden kann. Wenn es genutzt werden kann stellt sich die Frage, wie es genutzt werden kann? Zudem kommt die Frage auf, ob bestehende Technologien und Frameworks durch den Einsatz von WebAssembly überflüssig werden und aus der Technologielandschaft einer Website entfernt werden können? Diese Fragen werden im Folgenden beantwortet.

5.1 Allgemeine Informationen zur Oberflächengestaltung

WebAssembly besitzt keine Möglichkeit, direkt auf das Document Object Model (DOM) einer Website zuzugreifen. Das DOM stellt die Oberfläche der Website in einer Baumstruktur dar. Änderungen am DOM verändern die Oberfläche einer Website. Änderungen am DOM werden auch als Manipulationen bezeichnet. Kein direkter Zugriff aufs DOM bedeutet, dass WebAssembly nicht direkt genutzt werden kann, um Oberflächen zu manipulieren [MDN20e].

Über den Glue Code kann Wasm das DOM einer Website jedoch indirekt manipulieren. Dafür wird aus dem Wasm-Code eine JS-Funktion aufgerufen. In der JS-Funktion wird dann das DOM manipuliert. Die Manipulation mithilfe von JS wird als indirekte Manipulation bezeichnet [MDN20e]. In Kapitel 5.2 wird sie anhand eines Beispiels demonstriert.

Der Einsatz von WebAssembly in einer Webanwendung wird Frontend-Frameworks, wie das Frontend-CSS-Framework Bootstrap oder die JS-Frameworks Vue.js, Angular und React, nicht ersetzen. WebAssembly ist als performante Erweiterung, aber nicht als Ersatz, zu JavaScript und dem dazu bestehendem Ökosystem gedacht [MDN20e].

DOM-Zugriff aus WebAssembly zu implementieren ist aktuell nicht geplant [W3C20d]. Diese Entscheidung hat mehrere Gründe. Zum Einen reicht JS für eine Manipulation des DOM aus. Dabei arbeitet es hoch performant, da ein großer Teil der Manipulationen von der Browser-Engine übernommen wird. Die Browser-Engine selbst ist in stark optimiertem C++-Code geschrieben und arbeitet daher schnell. Ein weiterer Grund ist, dass Manipulationen bereits indirekt über JS möglich sind. Das senkt den Druck, eine weitere Möglichkeit zur Manipulation zu schaffen [Jen20].

5.2 Indirekte Manipulation des DOM durch WebAssembly

WebAssembly kann das DOM indirekt über JavaScript manipulieren. Dafür wird aus dem WebAssembly-Code eine JavaScript-Funktion aufgerufen, in der das DOM manipuliert wird.

Die indirekte Manipulation ist keine Performancebremse. Sie kann daher aus Sicht der Geschwindigkeit bedenkenlos verwendet werden. Aufrufe zwischen Wasm und JS sind ähnlich schnell wie Aufrufe zwischen JS und JS [Cla18a]. Die Manipulation des DOM aus einer JS-Funktion, die vom Wasm-Code aufgerufenen wird, findet in der gleichen Geschwindigkeit statt, wie die Manipulation des DOM aus einer JS-Funktion, die aus JS-Code aufgerufen wird.

Abbildung 24: Screenshot einer in WebAssembly berechneten Animation

WebAssembly kann aufgrund seiner Geschwindigkeit dazu genutzt werden, aufwändige Animationen zu berechnen und sie dann durch die indirekte Manipulation des DOMs anzuzeigen. Unter `https://grthor.github.io/wasm-canvas-demo/test.html` findet sich ein Beispiel für eine in WebAssembly berechnete und über JavaScript ausgegebene Animation. Abbildung 24 zeigt einen Screenshot dieser Animation. Die Animation zeigt einen Farbverlauf auf einem 400x400 Pixel großen Canvas. Für jeden Frame wird das Canvas neu gezeichnet. Dabei läuft die Animation, auf dem in Kapitel 3.2.2 beschriebenen Computer, abhängig vom Browser mit 60 FPS (Firefox) bis 250 FPS (Chrome und

Edge). Auf mobilen Endgeräten werden Bildwiederholfrequenzen von 200 FPS (Firefox) bis 250 FPS (Chrome und Edge) erreicht.

5.3 Fazit

Die Analyse der Möglichkeiten zur Oberflächengestaltung mit WebAssembly hat gezeigt, dass WebAssembly nicht direkt zur Gestaltung von Website-Oberflächen verwendet werden kann. Oberflächen können jedoch indirekt manipuliert werden. Bestehende Technologien und Frameworks in der Technologielandschaft einer Website werden durch den Einsatz von WebAssembly nicht überflüssig.

Die indirekte Manipulation von Oberflächen kann dazu genutzt werden, aufwändige Animationen in WebAssembly zu berechnen und im Browser darzustellen. Das Beispiel in Kapitel 5.2 hat gezeigt, dass die so berechneten Animationen mit hohen Bildwiederholfrequenzen ausgegeben werden können. Die Animationen erscheinen flüssig auf Computern und Smartphones. Diese Tatsache erlaubt es, solche Animationen in produktiven Webseiten einzusetzen.

6 Erkenntnisse aus der Implementierung der Beispiele

Dieses Kapitel befasst sich mit den Erkenntnissen, die während der Implementierung der in der verwendeten Beispiele erlangt wurden. Sie sind besonders interessant für die Implementierung späterer Webanwendungen mit WebAssembly.

Die gewonnenen Erkenntnisse stammen aus allen Bereiche, die mit der Programmierung, Einbindung und Verwendung von Webseiten mit WebAssembly im Zusammenhang stehen.

Viele der hier aufgelisteten Erfahrungen sind nicht in Tutorials im Internet zu finden. Sie ergeben sich aus der umfangreichen Verwendung von WebAssembly. Die für die Erfahrungen notwendige Tiefe wird in vielen Tutorials nicht erreicht, da diese häufig nur Hello-World-Programme betrachten.

In den folgenden Unterkapiteln werden die Erkenntnisse aus den verschiedenen Bereichen der Programmierung und wie sie erlangt wurden beschrieben.

6.1 Wahl der Toolchain

Für die Kompilierung von C-Code zu WebAssembly existieren zwei bekannte Toolchains. Die Emscripten-Toolchain und die Clang-LLVM-Toolchain. Eine Einführung in beide Toolchains wurde in Kapitel 2.4 gegeben.

Zu Beginn der Entwicklung einer neuen Webanwendung mit WebAssembly stellt sich die Frage, welche Toolchain für die Kompilierung genutzt werden soll. Diese Frage wird im Folgenden beantwortet.

Bevor die Toolchains verwendet werden können müssen, sie aufgesetzt werden. Anleitungen dazu gibt es im Internet viele. Häufig sind sie jedoch veraltet. Ein Grund dafür ist die hohe Geschwindigkeit, in der Teile der Toolchains geändert und neue Teile hinzugefügt werden. Das Problem der veralteten Anleitungen besteht bei beiden Toolchains. Im Vergleich sind die Anleitungen zur Emscripten-Toolchain jedoch etwas aktueller als die zur Clang-LLVM-Toolchain. Ein Zitat aus der offiziellen Dokumentation der Clang-LLVM-Toolchain („Getting started with the LLVM System“) beschreibt die Aktualität der Anleitungen:

> „The LLVM Getting Started documentation may be out of date.“ [LLV21]

Das Aufsetzen der Emscripten-Toolchain funktioniert aufgrund der aktuelleren Anleitungen schneller als das Aufsetzen der Clang-LLVM-Toolchain. Aus Sicht des Aufsetzens ist die Emscripten-Toolchain der Clang-LLVM-Toolchain vorzuziehen.

Eine Erkenntnis, die beim Aufsetzen gewonnen wurde, ist, dass nur die offiziellen Dokumentationen genutzt werden sollten. Das erleichtert das Aufsetzen, da man sich sicher sein kann, dass die Informationen stimmen, bzw. früher gestimmt haben. Die Korrektheit der Daten aus Anleitungen Dritter ist nur teilweise gegeben. Auch stellen die Anleitungen Dritter nicht immer die Best Practice dar.

Der Funktionsumfang beider Toolchains ist derselbe. Das ist nicht verwunderlich, da beide die gleiche IR (LLVMIR) und das gleiche Compiler-Backend (LLVM-WebAssembly-Backend) verwenden. Die Frontend-Compiler (emcc bzw. Clang) sind Drop-in Replacements für GCC und besitzen daher denselben Umfang. Aus Sicht des Funktionsumfangs ist es daher egal, ob die Emscripten- oder Clang-LLVM-Toolchain verwendet wird.

Nutzt der Code, der zu WebAssembly kompiliert werden soll, die C-Standard-Library (libc), so muss sie zu WebAssembly kompiliert werden. Das ist keine triviale Aufgabe. Warum die C-Standard-Library zu Wasm kompiliert werden muss, wird in Kapitel 6.3 erklärt. Wird die libs genutzt, so sollte die Emscripten-Toolchain genutzt werden. Sie fügt dem generierten WebAssembly-Code Teile einer bereits zu Wasm kompilierten libc hinzu. Im Zusammenspiel mit dem von Emscripten erzeugten Glue Code können so Funktionen aus der C-Standard-Library in WebAssembly genutzt werden. In der Clang-LLVM-Toolchain muss die libc selbst kompiliert und eigener Glue Code geschrieben werden. Das ist mit viel Aufwand verbunden.

Nutzt der Code, der zu WebAssembly kompiliert werden soll, die libc nicht, so ist es egal, welche Toolchain verwendet wird.

Die Verwendung der Emscripten-Toolchain bringt einen Nachteil mit sich. Sie erzeugt standardmäßig Glue Code. Der Glue Code ist nur grob auf den verwendeten C-Code zugeschnitten. Es kann daher passieren, dass zu viel Glue Code für die vorhandene Funktionalität des C-Codes erzeugt wird. Beim Aufruf einer Seite wird der möglicherweise überschüssige Glue Code heruntergeladen. Das verlangsamt die Ladegeschwindigkeit einer Webseite. Bei Verwendung der Clang-LLVM-Toolchain wird kein Glue Code erzeugt. Der Programmierer muss diesen selbst schreiben. Dadurch ist er wahrscheinlich genauer auf den C-Code zugeschnitten. Die Menge an Glue Code ist dadurch geringer und die Ladegeschwindigkeit einer Webseite höher. Einen Vergleich zwischen den Glue Code Mengen der Toolchains ist in Kapitel 3.2.3 gegeben. Steht die Erhöhung der Ladegeschwindigkeit im Fokus, daher sollte die Clang-LLVM-Toolchain genutzt werden.

6.2 Instantiierung von WebAssembly

Bevor WebAssembly-Code auf einer Webseite verwendet werden kann, muss er heruntergeladen, kompiliert und instantiieren werden. Abbildung 25 zeigt den zeitlichen Ablauf der drei Aufgaben. Die dafür benötigte Zeit bestimmt maßgeblich die Ladegeschwindigkeit einer Webanwendung. Es gibt mehrere Funktionen, die die drei Aufgaben erledigen. Sie erreichen alle dasselbe Ziel, die Erzeugung einer lauffähigen Instanz von WebAssembly. Die Funktionen sind jedoch unterschiedlich performant. Die Performance-Unterschiede sind deutlich zu spüren, weshalb die Instantiierung nur durch eine Funktion geschehen sollte.

Abbildung 25: Notwendige Aufgaben vor der Nutzung von WebAssembly

In Tutorials zu WebAssembly werden großteils zwei Funktionen zur Instantiierung verwendet:

1. `WebAssembly.instantiate()`
2. `WebAssembly.instantiateStreaming()`

Die zweite Funktion ist deutlich performanter als die erste Funktion. Daher sollte die Zweite verwendet werden. Bei Verwendung der ersten Funktion sind die drei Aufgaben (Herunterladen, Kompilieren und Instantiieren) jedoch besser ersichtlich. Daher wird im Folgenden zuerst die langsamere, aber nachvollziehbarere Funktion beschrieben. Anschließend wird die schnellere zweite Funktion betrachtet [Cla18b]. Dabei wird beschrieben, warum diese effizienter ist als die Erste. Abschließend werden zwei wichtige Erkenntnisse zur Implementierung der Instantiierung genannt.

Bei der Funktion `WebAssembly.instantiate()` wird der WebAssembly-Code wahlweise über `fetch()` oder ein XMLHttpRequest heruntergeladen. Der heruntergeladene Code wird in beiden Fällen in einem ArrayBuffer gespeichert. Erst nachdem der komplette Code heruntergeladen wurde, kann mit der Kompilierung und Instantiierung begonnen werden. Sie findet durch die Funktion `WebAssembly.instantiate()` statt [MDN20c]. Die Kompilierung und Instantiierung kann aber auch getrennt über `WebAssembly.compile()` und `WebAssembly.Instance()` erfolgen. Auf diese Funktionen wird hier aber nicht eingegangen, da sie für einen Sonderfall der Instanziierung gedacht sind, die Instanziierung eines gecachten Wasm-Moduls [MDN21b].

Listing 3 zeigt das Herunterladen, Kompilieren und Instantiieren über `WebAssembly.instantiate()`. Der Nachteil bei dieser Funktion ist, dass erst mit dem Kompilieren

und Instantiieren begonnen wird, nachdem der Wasm-Code komplett in einen Array-Buffer geladen wurde. WebAssembly ist jedoch so designt, dass Code bereits während des Herunterladens kompiliert werden kann. Das erhöht die Ladegeschwindigkeit einer Webanwendung stark [MDN20c].

```
fetch('module.wasm').then(response =>
        response.arrayBuffer()
).then(bytes =>
        WebAssembly.instantiate(bytes, importObject)
).then(results => {
        // Do something with the results!
});
```

Listing 3: Instantiierung über `WebAssembly.instantiate()` [MDN20c]

`WebAssembly.instantiateStreaming()` fasst das Herunterladen, Kompilieren und Instantiieren in einem Schritt zusammen. Während des Herunterladens wird Wasm-Code bereits kompiliert. Das ist möglich, da die Kompilierung direkt auf dem ankommenden Bytestrom stattfindet. Die Verwendung von `WebAssembly.instantiateStreaming()` zur Instantiierung wird in Listing 4 gezeigt. Sie ist deutlich schneller als `WebAssembly.instantiate()` [MDN20c].

```
WebAssembly.instantiateStreaming(fetch('simple.wasm'),
   importObject)
.then(results => {
        // Do something with the results!
});
```

Listing 4: Instantiierung über `WebAssembly.instantiateStreaming()` [MDN20c]

Bei der Implementierung der Beispiele sind zwei wichtige Erkenntnisse entstanden, die bei der Nutzung von WebAssembly beachtet werden sollten:

1. `WebAssembly.instantiateStreaming()` funktioniert nur, wenn der Server die .wasm-Dateien mit dem MIME-Type *application/wasm* bereitstelle. Nur aktuelle Server liefern den korrekten Typ. Es ist daher wichtig, einen aktuellen Server für eine Webanwendung mit WebAssembly zu nutzen. `WebAssembly.instantiate()` funktioniert auch mit einem falschen MIME-Type, bspw. *text/plain*.

2. Wird die Emscripten-Toolchain genutzt, so findet das Herunterladen, Kompilieren und Instantiieren automatisch im Glue Code statt. Der Emscripten Glue Code nutzt primär `WebAssembly.instantiateStreaming()`, besitzt jedoch eine Fallback-Option auf `WebAssembly.instantiate()`, wenn der falsche MIME-Type geliefert wird. Bei Verwendung der Clang-LLVM-Toolchain muss das Instantiieren selbst ausgelöst werden. Dafür kann der Code aus den Listings 3 oder 4 verwendet werden.

6.3 Hindernisse bei der Portierung vorhandener Codebasen zu WebAssembly

In diversen Artikeln im Internet finden sich Aussagen wie „... you can compile code written in other languages (e.g. C, C++ and Rust) to WebAssembly and run that code in the browser.“ [Abo19b] oder „Re-use existing code by targeting WebAssembly, embedded in a larger JavaScript / HTML application.“ [W3C20e]. Die Aussagen lassen vermuten, dass vorhandener C-Code unverändert zu WebAssembly kompiliert werden kann. Das ist eine verlockende Aussicht, da es bedeuten würde, dass dieselbe Codebasis zu einer Desktop- und einer Webanwendung kompiliert werden kann. Die Aussicht muss aber getrübt werden, da C-Code nur teilweise unverändert zu WebAssembly kompiliert werden kann. Das hat zwei Gründe:

1. Die Laufzeitumgebungen von C-Anwendungen und Webanwendungen unterscheiden sich. Die native Umgebung, in der C-Anwendungen herkömmlicherweise laufen, verfügt über mehr Funktionen, als die Browser-Umgebung, in der Webanwendungen laufen. Beispiele für Funktionen, die in der Browser-Umgebung nicht vorhanden sind, sind die Unterstützung von architekturspezifischem Inline-Assemblercode, einem Dateisystem oder SIMD.

   Bei Verwendung der Emscripten-Toolchain können bestimmte Funktionen, die eigentlich nicht in der Browser-Umgebung zur Verfügung stehen, benutzt werden. Emscripten emuliert die Funktionen im Glue Code. Bei Verwendung der Clang-LLVM-Toolchain können sie nicht standardmäßig genutzt werden. Die Emulation muss im Glue Code selbst geschrieben werden.

2. WebAssembly ist eine eigene Architektur wie x86, x86_64 oder arm. Die Architektur von WebAssembly heißt wasm32 [W3C20b]. Alle im C-Code verwendeten Libraries müssen zu wasm32 kompiliert werden. Die Libraries dürfen keine in Grund 1 genannten Funktionen nutzen, die in der Browser-Umgebung nicht zur Verfügung stehen. Diverse bekannte Libraries wie libc oder OpenSSL verwenden jedoch Funktionen, die in der Browser-Umgebung nicht zur Verfügung stehen. Sie können nicht oder nur teilweise zu WebAssembly kompiliert werden. Bei der Kompilierung von OpenSSL zu Wasm müssen beispielsweise der Hardwaresupport und Inline-Assemblercode deaktiviert werden [Apo19].

Die Aussicht, dass aller vorhandener C-Code ohne Änderungen zu WebAssembly kompiliert werden kann, ist aktuell aufgrund der beiden genannten Gründe nicht erfüllt. Alle Funktionen im C-Code, die nicht in der Browser-Umgebung zur Verfügung stehen, müssen vor der Kompilierung aus dem Code entfernt oder bei der Kompilierung deaktiviert werden. In Zukunft kann sich die Situation jedoch ändern. Ein Ziel von Emscripten ist es, dass C-Code ohne Änderungen zu WebAssembly kompiliert werden kann [Ems21].

Es gibt jedoch auch positive Szenarien, in denen vorhandener Code mit nur minimalen Änderungen zu WebAssembly kompiliert werden kann. Ein Beispiel dafür ist die Msieve-Faktorisierung in CrypTool Online (Abbildung 26) [Rec20a]. Dort wurde eine C-Bibliothek zur Faktorisierung großer Ganzzahlen zu WebAssembly kompiliert. Die einzige Änderung am Source Code war die Verlagerung der Ein- und Ausgabe weg von der Kommandozeile (da sie in Wasm nicht zur Verfügung steht) hin zur Website [Rec20b].

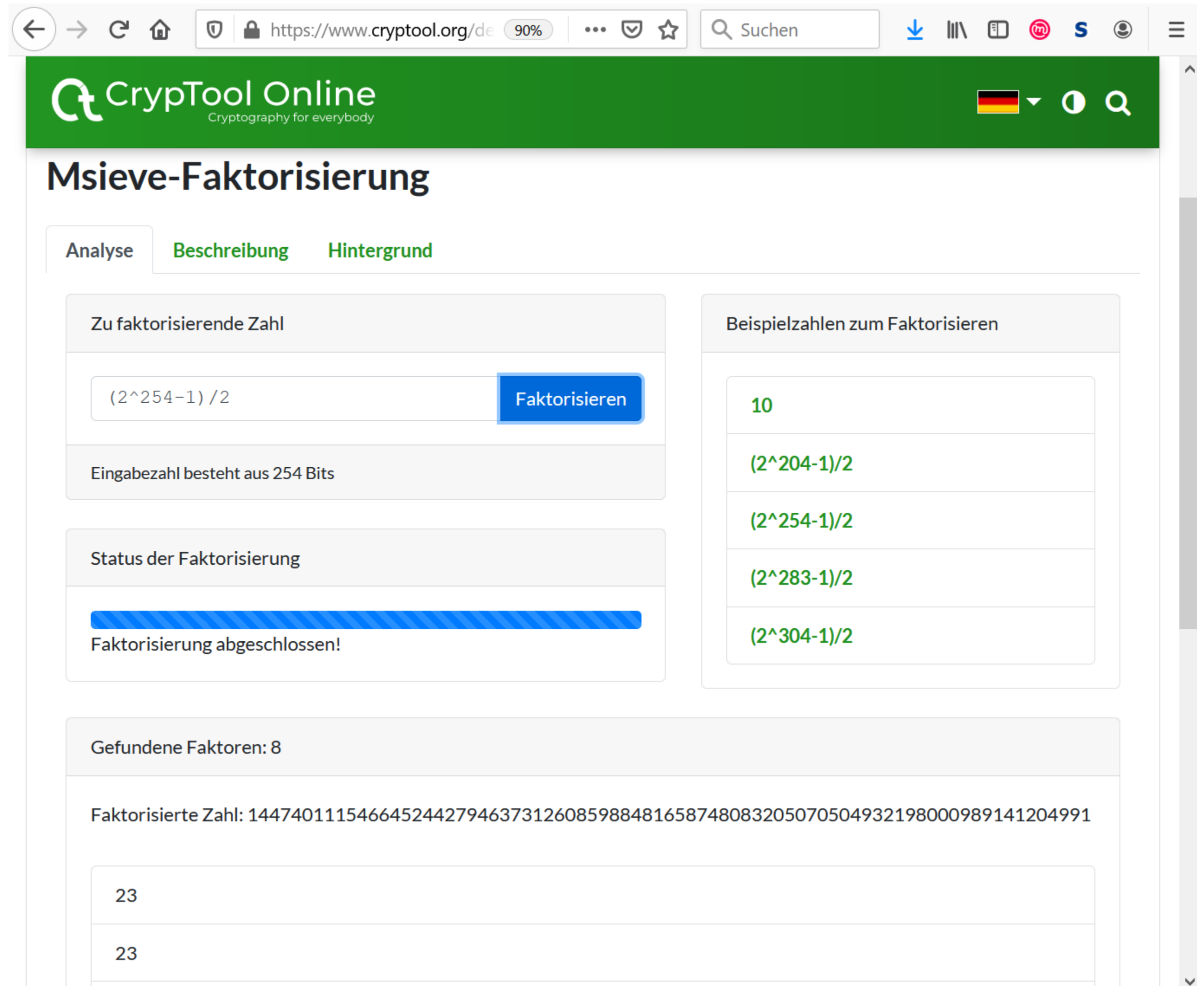


Abbildung 26: Screenshot der Msieve-Faktorisierung

6.4 Exception Handling

Exception Handling, die Behandlung von Ausnahmezuständen (engl. Exceptions), ist ein wichtiger Bestandteil moderner Programmiersprachen. Es verhindert das Abstürzen eines Programms bei Fehlern oder unerwarteten Zuständen. Exceptions sind ein Verfahren, bei dem Informationen über den Fehlerzustand an andere Programmebenen zur

Weiterbehandlung weitergereicht werden. Aus technischer Sicht stellen sie einen zweiten, optionalen Rückgabewert einer Methode dar [RA18].

WebAssembly besitzt kein Exception Handling. Jeder Fehler endet in einem Absturz des Programms. In mit der Emscripten-Toolchain kompilierten WebAssembly-Programmen gibt es jedoch Exception Handling [W3C20d].

Das Fehlen von Exception Handling in Wasm ist damit begründet, dass WebAssembly und sein Embedder, also die JSVM bei Webanwendungen oder Wasmtime bei Desktopanwendungen, unterschiedliche Auffassungen haben, was Exceptions sind. WebAssembly soll in verschiedenen Umgebungen laufen, weshalb es viele unterschiedliche Embedder gibt. Das stellt ein Problem dar, da WebAssembly und sein Embedder dieselbe Auffassung von Exceptions benötigen. Das ist notwendig, da Exceptions über unterschiedliche Programmebenen gereicht werden. Alle Ebenen müssen verstehen, dass eine Exception geworfen wurde. Das ist zum behandeln (bspw. schließen von Strömen / freigeben von Speicher) der Exception notwendig [Ahn21].

Wie bereits erwähnt, ist Exception Handling, in mit der Emscripten-Toolchain kompilierten Programmen, möglich. Emscripten emuliert das Exception Handling im Glue Code. Das funktioniert, indem vor der Ausführung einer möglicherweise exceptionwerfenden Funktion diverse Informationen durch JS gespeichert werden. Tritt eine Exception auf, kann anhand der gespeicherten Daten die Ausnahme behandelt und die Ausführung des Codes fortgesetzt werden. Das Speichern der Informationen in JS ist aufwändig und senkt die Geschwindigkeit des Codes. Daher ist es standardmäßig deaktiviert. Mit dem Compiler-Flag $-s\ DISABLE_EXCEPTION_CATCHING = 0$ kann es jedoch aktiviert werden [bri19].

Abbildung 27 zeigt einen Stacktrace nach einer unabgefangenen Exception in der Konsole von Firefox. Der Stacktrace ist mehr oder weniger nutzlos. Er zeigt nicht an, welche Exception geworfen wurde, sondern nur die Nummer „5247528“. Weiterhin hört er im JavaScript-Code auf. Welche Funktionen im WebAssembly-Code aufgerufen wurden ist nicht ersichtlich. Das in Abbildung 27 gezeigte Beispiel kann unter `https://grthor.github.io/C++_Exception_Handling/kaboom.html` selbst nachvollzogen werden.

```
exception thrown: 5247528                                                        kaboom.html:1246:19
    printErr https://grthor.github.io/C++_Exception_Handling/kaboom.html:1246
    callMain https://grthor.github.io/C++_Exception_Handling/kaboom.js:2529
    doRun    https://grthor.github.io/C++_Exception_Handling/kaboom.js:2579
    run      https://grthor.github.io/C++_Exception_Handling/kaboom.js:2590
Uncaught 5247528                                                                 kaboom.js:2533:5
    callMain https://grthor.github.io/C++_Exception_Handling/kaboom.js:2533
    doRun    https://grthor.github.io/C++_Exception_Handling/kaboom.js:2579
    run      https://grthor.github.io/C++_Exception_Handling/kaboom.js:2590
```

Abbildung 27: Stacktrace nach einer Exception in Wasm

Zusammenfassend ist Exception Handling in purem WebAssembly-Code nicht möglich. Emscripten erlaubt es jedoch, indem das Exception Handling im Glue Code emuliert wird. Das reduziert jedoch die Geschwindigkeit einer WebAssembly-Anwendung, weshalb es standardmäßig deaktiviert ist. Aufgrund des Einflusses auf die Geschwindigkeit einer Anwendung sollte es in produktiven Anwendungen deaktiviert bleiben.

6.5 Speicherreservierung und -nutzung

Bei der Verwendung von WebAssembly müssen häufig Daten zwischen JavaScript und WebAssembly ausgetauscht werden. Ein möglicher Anwendungsfall dafür ist, dass ein Nutzer Werte auf einer Website eingibt und diese dann von WebAssembly verarbeitet werden. Damit die Werte verarbeitet werden können, müssen sie im JavaScript-Code erfasst und an Wasm übergeben werden. Wasm kann die Werte nicht selbst erfassen, da es keinen Zugriff auf das DOM hat. Für die Übergabe müssen die Werte zwischengespeichert werden. Die Reservierung und Nutzung besitzt einige Tücken. Sie wird daher im Folgenden vorgestellt wird.

JavaScript und WebAssembly nutzen denselben Speicher. Das ist die Basis dafür, dass Daten überhaupt ausgetauscht werden können. Die Nutzung desselben Speichers ist nicht selbstverständlich, da JS-Code in einer Sandbox ausgeführt wird. Der in der Sandbox verwendete Speicher steht nur dem JavaScript-Code zur Verfügung. Hier kommt die Tatsache zum Tragen, dass der WebAssembly-Code im JavaScript-Code instantiiert wird. Er läuft, vereinfacht ausgedrückt, im JavaScript-Code. Daher haben JS und Wasm Zugriff auf denselben Speicher.

Zur Erzeugung einer Wasm-Instanz wird ein Speicherobjekt benötigt. Das Speicherobjekt wird im JavaScript-Code erzeugt und definiert die initiale und maximale Größe des Speichers zwischen JS und Wasm. Der Speicher kann innerhalb dieser Spanne seitenweise vergrößert werden. Eine Seite ist 64 KiB groß. Die maximale Speichergröße ist 4 GiB [HKZ20]. Der Speicher kann aus Sicht des JavaScript-Codes als ArrayBuffer gesehen werden. Aus Sicht des WebAssembly-Codes kann er als Arbeitsspeicher betrachtet werden [MDN20d].

Die Art der Reservierung von Speicher unterscheidet sich je nach verwendeter Toolchain. Wird die Emscripten-Toolchain genutzt, so werden die C-Funktionen `malloc` und `calloc` im Glue Code nachgebildet. Über sie kann aus dem JS- und Wasm-Code heraus Speicher allokiert werden. Als Rückgabewert liefern beide Funktionen einen Pointer auf das erste Element im reservierten Speicherbereich. In der Clang-LLVM-Toolchain gibt es diese Funktionen nicht. Das heißt, bei der Programmierung müssen die Pointer auf freie Speicherbereiche selbst gesetzt werden. Das wiederum bedeutet, dass der Programmierer beim Ablegen von Daten im Speicher schauen muss, in welchen Bereichen bereits Daten liegen und welche Bereiche frei sind. Das erschwert die Programmierung. Eine

Erkenntnis daraus ist, dass die Programmierung einer Website mit WebAssembly einfacher ist, wenn man die Emscripten-Toolchain nutzt, da man die Speicherreservierung von Emscripten nutzen kann.

Unabhängig von der verwendeten Toolchain ist der Speicher ein linearer Speicher. Linear bedeutet, dass es ein großer zusammenhängender Speicherbereich ist. Der Speicher ist als ArrayBuffer in JS implementiert. Für die Nutzung aus JS heraus wird über den Speicherbereich ein Typed Array gelegt. Das Typed Array erlaubt das Lesen und Schreiben von rohen Bytes in den ArrayBuffer. Es können mehrere sogenannte Typed Array Views auf denselben ArrayBuffer gelegt werden [MDN21a]. In Abbildung 28 sind vier Typed Array Views, `Uint8Array`, `Uint16Array`, `Uint32Array` und `Float64Array` über einen 16 Byte langen ArrayBuffer gelegt.

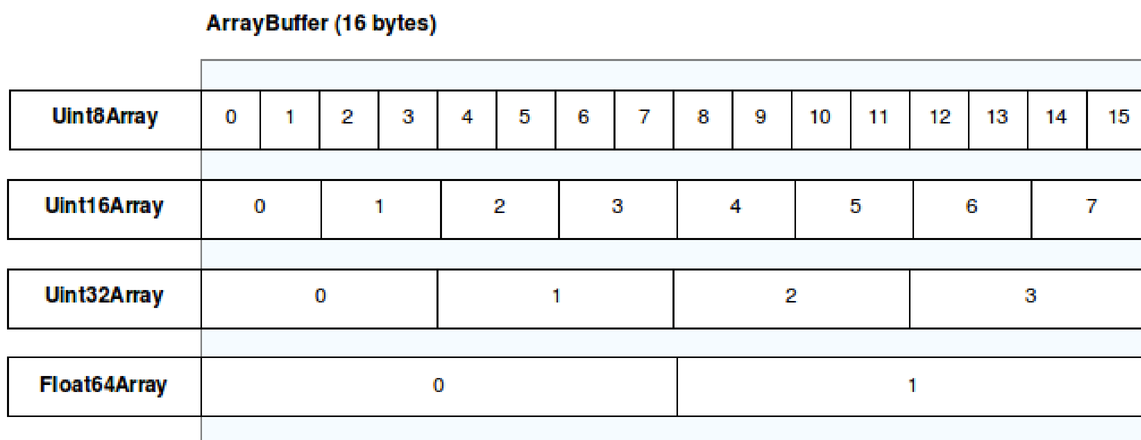


Abbildung 28: ArrayBuffer mit mehreren Typed Array Views [MDN21a]

Die Adressierung von Objekten im ArrayBuffer findet in JS über die Indizes der Typed Array Views statt. Sollen beispielsweise zwei 32 Bit Integers gespeichert werden, so würden sie bei einem leeren ArrayBuffer an den Stellen `Uint32Array[0]` und `Uint32Array[1]` gespeichert. Die Adressierung von Objekten aus WebAssembly heraus findet dagegen byteweise statt. In Wasm würde der Wert von `Uint32Array[0]` an der Stelle 0 beginnen, der Wert von `Uint32Array[1]` an der Stelle 4. Das hat den Grund, dass der zweite 32 Bit Integer beim 5. Byte beginnt. Die ersten 4 Byte sind mit dem ersten 32 Bit Integer belegt. Die unterschiedliche Adressierung erschwert die Übergabe von Daten zwischen WebAssembly und JavaScript.

Der Speicher kann zur Laufzeit vergrößert werden. Der Befehl zum Vergrößern des Speichers ist `WebAssembly.Memory.prototype.grow()`. Der Speicher wird seitenweise vergrößert. Eine Seite ist 64 KiB groß. Die maximale Speichergröße ist 65536 Seiten. Das entspricht 4 GiB. 4 GiB ist die maximal in WebAssembly verwendbare Speichergröße, da über einen 32 Bit Integer nicht mehr Speicheradressen abgebildet werden können

[HKZ20]. `WebAssembly.Memory.prototype.grow()` muss bei Verwendung der Clang-LLVM-Toolchain vor der Reservierung des von Speicher aufgerufen werden. Bei Verwendung der Emscripten-Toolchain verbirgt sich der Befehl hinter `malloc` und `calloc`.

Wird die Emscripten-Toolchain verwendet, so muss reservierter Speicher über `free` freigegeben werden. `free` gibt keinen realen Speicher frei. Es gibt ihn nur im Emscripten Glue Code frei. Nutzt man `free` nicht, reserviert Emscripten bei jedem Aufruf von `malloc` oder `calloc` neuen, ungenutzten Speicher. Wird `free` genutzt, so wird bei einer Reservierung freigegebener Speicher genutzt. Die Wiederverwendung von freigegebenem Speicher senkt den Speicherbedarf einer Anwendung. Der Speicher wird real freigegeben, wenn die Website verlassen wird. Der JSVM Garbage Collector erkennt in diesem Fall, dass der ArrayBuffer nicht mehr genutzt wird und entfernt ihn aus dem Speicher [Cla17d].

6.6 Optimierung von JS-Code als Alternative zu WebAssembly

Die Nutzung von WebAssembly ist mit Aufwand verbunden. Bevor Wasm in einer Webanwendung verwendet werden kann, muss Wissen über die Programmierung mit WebAssembly gesammelt, eine Toolchain zur Kompilierung des Codes aufgesetzt und der erzeugte Code in eine bestehende Anwendung eingebettet werden. Die drei Schritte benötigen Zeit. Bei einer Entscheidung, ob WebAssembly zur Steigerung der Geschwindigkeit einer Webanwendung eingesetzt werden soll, sollte daher abgewogen werden, ob sich der Zeiteinsatz rentiert. Die Geschwindigkeit kann nämlich auch durch die Optimierung von bestehendem JS-Code gesteigert werden. Es gibt optimierten JS-Code, der mit der Geschwindigkeit von Wasm-Code mithalten kann.

Das in Kapitel 3.2 verwendete RC4-Beispiel zur Messung der Geschwindigkeit der Verschlüsselung in JavaScript wurde händisch optimiert. Die optimierte Implementierung ist unter folgendem Link zu finden: https://grthor.github.io/RC4_JS_geringe_Interaktion_optimiert/Grundgeruest.html. Abbildung 29 zeigt die Laufzeiten der RC4-Verschlüsselungen. Es ist zu erkennen, dass die herkömmliche JavaScript-Implementierung die längste Laufzeit (3329 ms) hat. Die Laufzeiten der optimierten JavaScript-Implementierung (685,6 ms) und der WebAssembly-Implementierungen (646,4 ms und 475,8 ms) liegen nah beieinander. Die optimierte JS-Implementierung ist nur 39,2 ms (6,1 %) langsamer als die Emscripten-WebAssembly-Implementierung und 209,8 ms (44,1 %) langsamer als die Clang-LLVM-WebAssembly-Implementierung.

Die Steigerung der Geschwindigkeit wurde durch drei Änderungen am Code erreicht. Zuerst wurde der JS-Code typisiert. Das heißt, es wurden anstelle der JavaScript-typischen, typunsicheren Variablendeklaration eine typsichere Deklaration verwendet. Die JSVM kann den Code dadurch besser optimieren [Cla17c]. Als nächstes wurden Arrays mit fester Länge für die Speicherung des Schlüssels, Klartextes und Geheimtextes Arrays ver-

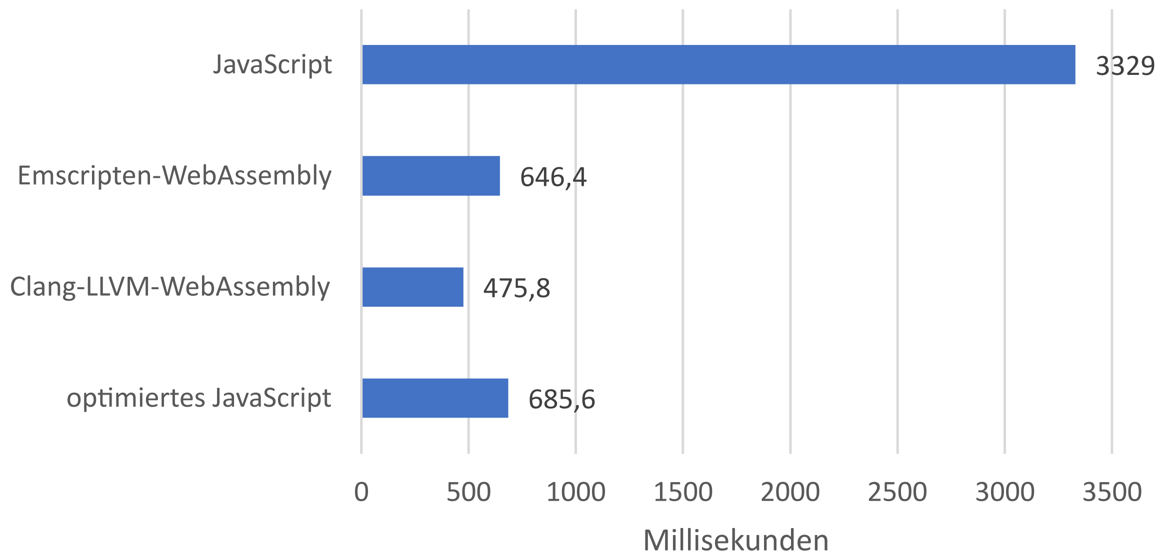


Abbildung 29: Optimierter JS-Code im Vergleich zu unoptimiertem JS-Code

wendet. Die Felder im Array wurden über ihre Indizes angesprochen. Der inperformante `Array.prototype.push()`-Befehl wurde so vermieden [MDN20d]. Zuletzt wurden die Datentypen an die auftretenden Werte angepasst. Es wurden 1-Byte-Integers anstelle von 4-Byte-Integers verwendet. Da sich alle Werte im Bereich von 0-255 befinden, ist das möglich. Die Anpassung der Datentypen reduziert den Speicherbedarf auf eine Viertel des zuvor benötigten Speichers. Dadurch wird weniger Zeit für die Allokierung von Speicherplatz benötigt [Cam19].

Die Erkenntnis aus diesem Kapitel ist, dass bei einer Entscheidung, ob Wasm zur Steigerung der Ausführungsgeschwindigkeit einer Webanwendung eingesetzt werden soll, bedacht werden sollte, dass vorhandener JS-Code optimiert werden kann. Die Optimierung könnte weniger Zeit in Anspruch nehmen, als die Neuimplementierung der Funktionalität in WebAssembly.

6.7 Geschwindigkeit von WebAssembly in verschiedenen Browsern

Die Geschwindigkeit von WebAssembly variiert je nach Browser. Die Unterschiede sind teils essentiell. Die Abbildungen 30 und 31 zeigen die Laufzeiten des RC4-Beispiels in mehreren Desktop- und Smartphone-Browsern. Die Desktop-Browser laufen auf dem in Kapitel 3.2.2 beschriebenen Rechner. Die Smartphone-Browser laufen auf einem Huawei

P20 Smartphone mit einem 8-Kern-Prozessor und 4 GB Arbeitsspeicher. Die verwendeten Browser-Versionen sind die zum Zeitpunkt der Erstellung des Vergleichs aktuellsten Versionen. Beide Abbildungen stellen die Zeiten zur Verschlüsselung eines 64 MiB langen Klartextes dar. Die Unterschiede zwischen den Laufzeiten und somit der Ausführungsgeschwindigkeiten der Implementierungen sind deutlich zu erkennen.

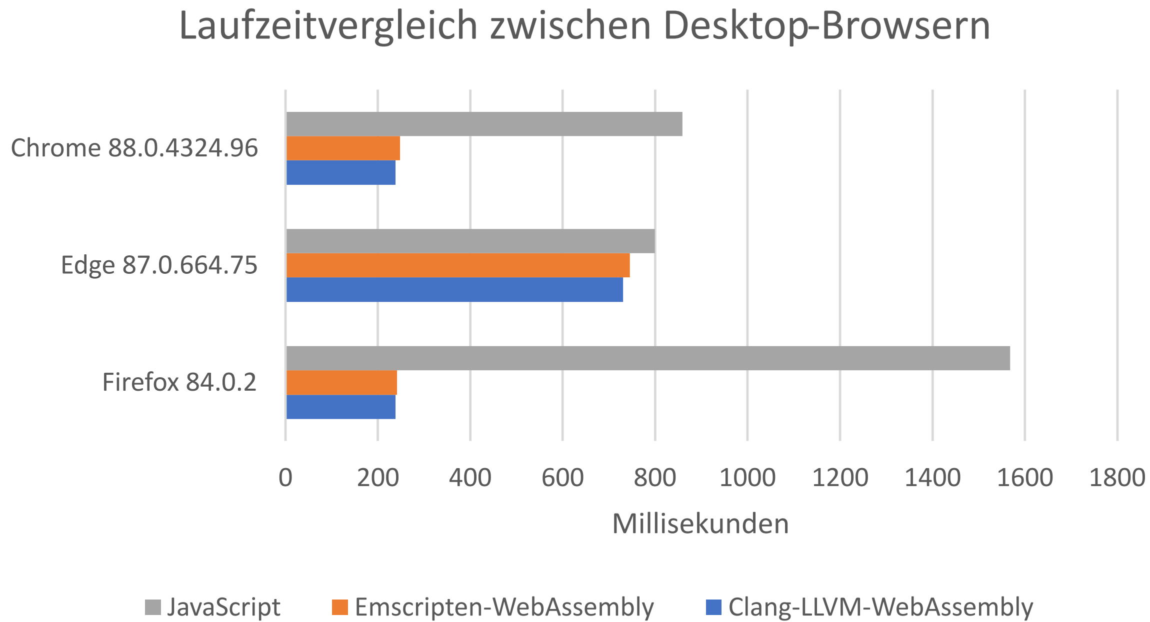


Abbildung 30: Vergleich der Laufzeiten zwischen unterschiedlichen Desktop-Browsern

Die Abbildungen 30 und 31 zeigen, dass die Verwendung von WebAssembly in allen getesteten Browsern zu einer Erhöhung der Ausführungsgeschwindigkeit führt. Die Geschwindigkeitssteigerungen fallen jedoch unterschiedlich stark aus. Insbesondere in den Desktop-Versionen von Chrome und Firefox, sowie den Mobilen-Versionen von Edge und Firefox, sind große Steigerungen erkennbar. Die mobile Version von Chrome ist im Vergleich zu den anderen beiden mobilen Versionen bereits bei der Ausführung des JS-Codes schnell. Die Geschwindigkeitssteigerung durch die Verwendung von WebAssembly fällt daher nicht so deutlich aus wie bei Edge und Firefox. Bei der Desktop-Version von Edge ist auffällig, dass die Geschwindigkeit des RC4-Beispiels durch die Verwendung von WebAssembly kaum steigt.

Aus diesem Kapitel lassen sich mehrere Erkenntnisse gewinnen. Die erste Erkenntnis ist, dass die Verwendung von WebAssembly, zumindest beim RC4-Beispiel, in allen Browsern zu einer Steigerung der Ausführungsgeschwindigkeit führt. Die zweite Erkenntnis ist, dass bei der Verwendung von WebAssembly zur Nutzung von Chrome oder Firefox geraten werden sollte. Das Nutzungserlebnis ist in beiden Browsern besser, da die Laufzeiten geringer und die Ausführungsgeschwindigkeiten höher sind.

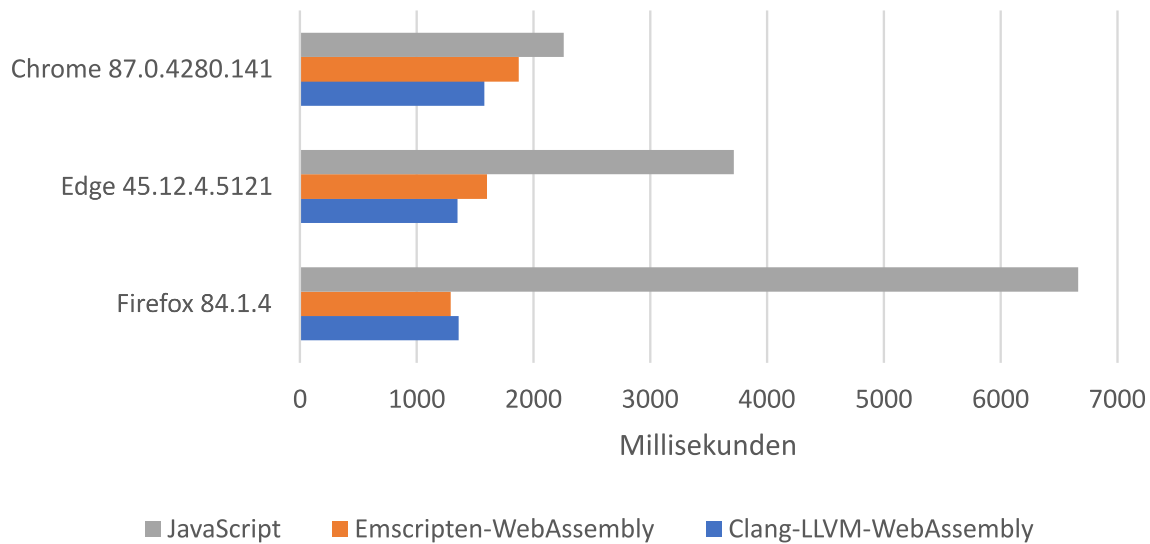


Abbildung 31: Vergleich der Laufzeiten zwischen unterschiedlichen Smartphone-Browsern

6.8 Fazit

Bei der Implementierung der Beispiele in dieser Arbeit konnten wichtige Erkenntnisse über die Entwicklung, Einbettung und Verwendung von WebAssembly in Webanwendungen gewonnen werden. Die Erkenntnisse vereinfachen Entscheidungen für oder gegen bestimmte Vorgehensweisen. Das verkürzt die Entwicklungszeit von Webanwendungen mit WebAssembly. Die in den vorherigen Kapiteln gesammelten Erkenntnisse werden im Folgenden zusammengefasst.

Die Wahl der Toolchain ist eine der ersten Fragen, die sich im Entwicklungsprozess stellt. Basierend auf den Erkenntnissen ist zur Verwendung der Emscripten-Toolchain zu raten. Sie ist besser dokumentiert als die Clang-LLVM-Toolchain. Zudem emuliert sie Teile der libc im Glue Code. Das vereinfacht die Portierung vorhandener Codebasen zu WebAssembly. Änderungen am Code beschränken sich dadurch nur auf Funktionen, die nicht in Wasm zur Verfügung stehen, u.a. SIMD und Inline-Assemblercode. Durch die Emulation der libc stehen die Befehle `malloc()` und `calloc()` im Browser zur Verfügung. Sie vereinfachen die Arbeit mit dem Speicher, indem sie Entwickler vom Tracken der freien und belegten Speicherbereiche befreien. Code kann dadurch schneller und fehlerfreier entwickelt werden.

Nachdem Code mit Hilfe einer Toolchain zu WebAssembly kompiliert wurde, stellt sich die Frage, wie der WebAssembly-Code instantiiert werden soll. Dazu kann die Funktion `WebAssembly.instantiateStreaming()` empfohlen werden. Sie ist deutlich schneller als andere Funktionen, indem sie Code bereits während des Downloads kompiliert. Dadurch ist die Ladezeit geringer und das Nutzungserlebnis besser. Es muss jedoch darauf geachtete werden, dass der Server die .wasm-Dateien mit dem MIME-Type *application/wasm* bereitstellt. Ansonsten ist die Funktion `instantiateStreaming()` nicht nutzbar. Wurde die Emscripten-Toolchain genutzt übernimmt der Glue Code die Instantiierung des Codes. Ein Programmierer muss sich dann nicht mehr darum kümmern.

Besteht das Ziel, eine vorhandene Codebasis zu WebAssembly zu kompilieren, muss beachtet werden, dass die Kompilierung nicht immer ohne Weiteres möglich ist. Alle im Code verwendeten Funktionen müssen im Browser zur Verfügung stehen. Ansonsten ist eine Kompilierung nicht möglich. Die Emscripten-Toolchain vereinfacht die Portierung. Sie bildet Funktionen aus dem nativen Umfeld im Browser nach. Für eine zügige Portierung sollte daher die Emscripten-Toolchain genutzt werden.

Exception Handling erhöht die Stabilität von Code. In purem WebAssembly-Code ist Exception Handling jedoch nicht möglich. In mit der Emscripten-Toolchain erzeugtem Code wird es simuliert und ist daher trotzdem möglich. Die Simulation reduziert jedoch die Geschwindigkeit, weshalb es standardmäßig deaktiviert ist. Aufgrund des Einflusses auf die Geschwindigkeit sollte es in produktiven Anwendungen deaktiviert bleiben.

Bezogen auf die Arbeit mit dem Speicher vereinfacht die Emscripten-Toolchain die Arbeit eines Programmierers. Erstens muss der Programmierer nicht selbst prüfen, welche Teile des Speichers bereits gefüllt sind und welche nicht. Zweitens kann Speicher aus dem JS- und Wasm-Code heraus reserviert werden. Punkt Eins reduziert die Fehleranfälligkeit des Codes deutlich. Punkt Zwei erleichtert die Programmierung, indem aus C bekannte Funktionen in Wasm verwendet werden können. Aus Sicht der Speicherreservierung und Nutzung sollte die Emscripten-Toolchain der Clang-LLVM-Toolchain vorgezogen werden.

WebAssembly kann zur Steigerung der Geschwindigkeit einer Webanwendung eingesetzt werden. Bei einer Entscheidung, ob die Geschwindigkeit durch den Einsatz von WebAssembly erhöht werden soll, sollte beachtet werden, dass auch JavaScript hohe Geschwindigkeiten erreichen kann. Der JS-Code muss dafür optimiert werden. Das Optimieren kann weniger Zeit in Anspruch nehmen, als das Ersetzen von JS-Code durch Wasm-Code.

Die letzte Erkenntnis aus dem Kapitel zur Implementierung der Beispiele ist, dass die Geschwindigkeit von WebAssembly abhängig vom Browser variiert. In allen Testfällen hat WebAssembly zu einer Steigerung der Geschwindigkeit im Vergleich zu JS geführt.

In Edge war die Steigerung jedoch deutlich geringer als in Firefox und Chrome. Daher sollte zur Verwendung von Firefox und Chrome geraten werden.

7 Related Work

Hass et. al. von Mozilla, Google, Apple und Microsoft stellen in ihrem Paper „Bringing the Web up to Speed with WebAssembly“ WebAssembly grundlegend dar [Haa+17]. Das Paper wird in allen Bereichen der Arbeit referenziert. Es gibt einen guten Überblick über WebAssembly und sein Design. Zusätzlich wird Wasm von seinen Vorgängern abgegrenzt, die Motivation für die Entwicklung beschrieben und auf die Geschwindigkeit und Sicherheit eingegangen.

Alon Zakai (@kripken) von Google, der Erfinder von Emscripten und Mit-Erfinder von WebAssembly, stellt in diversen Vorträgen und Posts die Emscripten- und Clang-LLVM-Toolchain vor. Er beschreibt die Funktion und die darin enthaltenen Tools [Zak15; Zak16; Zak18; Zak19]. Seine Beiträge werden hauptsächlich in der Vorstellung der WebAssembly-Toolchains referenziert.

Lin Clark von Mozilla erklärt in ihren mit Cartoons untermalten Berichten anschaulich Verfahren und Techniken, die die Geschwindigkeit von WebAssembly beeinflussen [Cla17a; Cla17b; Cla17c; Cla17d; Cla17e; Cla17f; Cla18a; Cla18b]. Sie stellt dabei komplexe Verfahren, wie Streaming Compilation, JIT-Kompilierung und Interop-Aufrufe, verständlich dar. Viele Informationen aus ihren Berichten sind in die theoretischen Untersuchung der Geschwindigkeit von WebAssembly eingeflossen.

Die Sicherheit von WebAssembly im Allgemeinen und die Speichersicherheit im Speziellen wird von Bergbom [Ber18a; Ber18b], Lehman et. al. [LKP20] und Musch et. al. [Mus+19b] betrachtet. Sie beschreiben die Sicherheit von WebAssembly und geben Beispiele, wie bekannte Schwachstellen ausgenutzt werden können. Aufbauend auf den Beispielen wurde der in der Arbeit gezeigte Code entwickelt.

Musch et. al. von der TU Braunschweig betrachten in ihren Papern [Mus+19a; Mus+19b] missbräuchliche Anwendungsgebiete von WebAssembly. Die Anwendungsgebiete werden erklärt und es wird eine Studie zur Verbreitung dieser gezeigt. Die Kapitel 4.4 und 4.5 basieren großteils auf diesen beiden Papern.

In die Analyse der Möglichkeiten zur Oberflächengestaltung ist hauptsächlich die Dokumentation von WebAssembly des MDNs [MDN20e] eingeflossen. Sie stellt prägnant dar, dass mit WebAssembly Oberflächen nur indirekt gestaltet werden können.

Neben den zuvor genannten Quellen wurden noch diverse weitere Quellen genutzt. Beispiele dafür sind die Dokumentationen von JavaScript und WebAssembly vom MDN und W3C, die Dokumentationen zu Emscripten, Clang und LLVM, sowie diverse Tutorials zu WebAssembly. Sie sind über die Arbeit verteilt eingeflossen, werden hier aber nicht explizit aufgelistet. Sie haben jedoch einen großen Teil zum Verständnis von WebAssembly beigetragen.

8 Zusammenfassung und Ausblick

Der neue Webstandard WebAssembly wurde analysiert und hinsichtlich seines praktischen Nutzens in unterschiedlichen Bereichen bewertet. Dazu wurden zu Beginn der Arbeit die Grundlagen von WebAssembly dargestellt. Anschließend wurde WebAssembly in den Bereichen Geschwindigkeit, Sicherheit und Oberflächengestaltung evaluiert. Dabei wurden Beispiele zum Nachvollziehen und Überprüfen des theoretisch angeeigneten Wissens erstellt. Bei der Implementierung der Beispiele sind Erkenntnisse über die Programmierung mit Wasm sichtbar geworden. Sie wurden gesammelt und in aufbereiteter Form dargestellt. Aus den Darstellungen heraus wurden Empfehlungen für die Programmierung mit Wasm abgeleitet. Damit wurden die Ziele 1 bis 5 aus Kapitel 1.2 erreicht. Ziel 6, die abschließende, bereichsübergreifende Bewertung, ob WebAssembly für den praktischen Einsatz geeignet ist, wird in diesem Kapitel erfüllt.

Die Evaluierung der Geschwindigkeit kam zu einer klaren Empfehlung zur Verwendung von WebAssembly. Wasm-Code wird deutlich schneller ausgeführt als JS-Code. Dagegen sind die Ladegeschwindigkeiten von JS-Code geringer. Sie werden durch die höheren Ausführungsgeschwindigkeiten jedoch mehr als ausgeglichen.

Die Evaluierung der Sicherheit von WebAssembly ergab, dass Wasm aus sicherheitstechnischen Aspekten verwendet werden kann. Es ist, was das Verlassen der Browserumgebung betrifft, so sicher wie JavaScript. Auf eine korrekte Eingabevalidierung muss jedoch geachtet werden. Findet sie nicht statt, sind Buffer Overflows und dadurch Manipulationen des Control Flows möglich.

Die Evaluierung der Möglichkeiten zur Oberflächengestaltung hat gezeigt, dass WebAssembly indirekt zur Gestaltung bzw. Manipulation von Weboberflächen eingesetzt werden kann. Es besitzt jedoch keine nativen Möglichkeiten dazu. Die Evaluation kommt daher zu dem Schluss, dass WebAssembly nicht primär zur Gestaltung von Weboberflächen genutzt werden sollte. Änderungen an der Benutzeroberfläche aus Wasm heraus sind aber möglich.

Durch die Implementierung der Beispiele ergaben sich mehrere wichtige Erkenntnisse, die die Bewertung von WebAssembly für den praktischen Einsatz beeinflussen. Es zeigte sich, dass bestehende Codebasen in vielen Fällen nicht ohne weiteres zu WebAssembly kompiliert werden können. Die Nutzung der Emscripten-Toolchain vereinfacht die Kompilierung jedoch. Auch ergab sich, dass es kein Exception Handling in WebAssembly gibt. Weiterhin beeinflusst die Art der Instantiierung die Ladegeschwindigkeit erheblich. Auf die korrekte Instantiierung muss daher geachtet werden. Eine weitere Hürde bei der Arbeit mit Wasm ist die Speichernutzung. Sie ist aufgrund des fehlenden Speichermanagements trickreich. Für die abschließende Bewertung bezüglich der Implementierung ergibt sich, dass die Nutzung von WebAssembly mit Schwierigkeiten verbunden ist. Daher kann keine uneingeschränkte Empfehlung aus Sicht der Programmierung zur Nut-

zung von WebAssembly gegeben werden. Die Schwierigkeiten sind jedoch lösbar oder durch Workarounds umgehbar.

Zusammenfassend kann auf Basis der drei Evaluierungen und den Erkenntnissen aus der Implementierung eine Empfehlung zur Nutzung von WebAssembly in Webseiten gegeben werden. Es bietet das Potenzial, bei rechenintensiven Aufgaben die Geschwindigkeit einer Website deutlich zu steigern. Dabei wird der Code in einer sicheren Umgebung ausgeführt und es besteht die Möglichkeit, Oberflächen zu gestalten. Leicht nachteilig wirken sich die Schwierigkeiten bei der Programmierung mit WebAssembly auf die Empfehlung aus.

Der Webstandard WebAssembly wurde in der Arbeit grundlegend analysiert. Die Entwicklung des Standards ist jedoch nicht abgeschlossen, sondern schreitet kontinuierlich voran. Es sind diverse Features in Planung, welche die Geschwindigkeit, Sicherheit und Programmierung mit WebAssembly beeinflussen werden. Daher sollte eine Evaluierung des Standards in Zukunft erneut durchgeführt werden. Insbesondere neue Features im Bereich Geschwindigkeit und Sicherheit sollten evaluiert werden.

Die Oberflächengestaltung wurde im Wesentlichen analysiert. Es wurde anhand einer Animation gezeigt, dass aufwändige Berechnungen in WebAssembly getätigt und auf einer Webseite visualisiert werden können. Die Interaktion zwischen JavaScript und WebAssembly, sowie die Ausgabe im Browser kann optimiert werden. Der Einsatz von WebGL in Zusammenarbeit mit WebAssembly ist in diesem Szenario besonders interessant. Die hardwarebeschleunigte Darstellung von Grafiken und Animationen im Browser kann zu einer Steigerung der Frame Rates und somit zu einer flüssigeren Wiedergabe führen. Daher sollten die Möglichkeiten zur Nutzung von WebGL und WebAssembly in einer weiterführenden Arbeit untersucht werden.

Literaturverzeichnis

[Abo19a] Robert Aboukhalil. *How We Used WebAssembly To Speed Up Our Web App By 20X (Case Study)*. 2019. URL: https://www.smashingmagazine.com/2019/04/webassembly-speed-web-app/ (besucht am 27. 11. 2020).

[Abo19b] Robert Aboukhalil. *Porting Games to the Web with WebAssembly*. 2019. URL: https://medium.com/@robaboukhalil/porting-games-to-the-web-with-webassembly-70d598e1a3ec (besucht am 21. 01. 2021).

[Ahn21] Heejin Ahn. *Exception handling*. 2021. URL: https://github.com/WebAssembly/exception-handling/blob/master/proposals/exception-handling/Exceptions.md (besucht am 26. 02. 2021).

[Aka17] Akamai Technologies Inc. *Akamai Online Retail Performance Report: Milliseconds Are Critical*. 2017. URL: https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp (besucht am 27. 11. 2020).

[Apo19] Richard L. Apodaca. *Compiling C to WebAssembly and Running It - without Emscripten*. 2019. URL: https://depth-first.com/articles/2019/10/16/compiling-c-to-webassembly-and-running-it-without-emscripten/ (besucht am 22. 02. 2021).

[App20] Apple Inc. *A fast, open source web browser engine*. 2020. URL: https://webkit.org/ (besucht am 13. 10. 2020).

[Ber18a] John Bergbom. *Memory safety: old vulnerabilities become new with WebAssembly*. Forcepoint LLC. 2018. URL: https://www.forcepoint.com/sites/default/files/resources/files/report-web-assembly-memory-safety-en.pdf (besucht am 25. 01. 2021).

[Ber18b] John Bergbom. *WebAssembly security: potentials and pitfalls*. Forcepoint LLC. 2018. URL: https://www.forcepoint.com/blog/x-labs/webassembly-potentials-and-pitfalls (besucht am 25. 01. 2021).

[Bit17] BitcoinMag-Presseteam. *Geld verdienen: Coinhive bietet neue Möglichkeiten der Website-Monetarisierung*. 2017. URL: https://www.bitcoinmag.de/news/coinhive-bietet-neue-art-der-website-monetarisierung (besucht am 17. 02. 2021).

[Bli18] Blindman67. *Why is webAssembly function almost 300 time slower than same JS function*. 2018. URL: https://stackoverflow.com/questions/48173979/why-is-webassembly-function-almost-300-time-slower-than-same-js-function (besucht am 27. 11. 2020).

[bri19] brionv. *Exception handling in emscripten: how it works and why it's disabled by default*. 2019. URL: https://brionv.com/log/2019/10/24/exception-handling-in-emscripten-how-it-works-and-why-its-disabled-by-default/ (besucht am 26. 02. 2021).

[Cal18] Dan Callahan. *Approaching WebAssembly*. Mozilla Foundation. 2018. URL: https://storage.eventyay.com/eventyay.com/events/temp/248a87d8-30b1-4bae-90af-6bd15f7b3b53/cVpFQm1nWl/Wasm.pdf (besucht am 19. 10. 2020).

[Cam19] Bret Cameron. *13 Tips to Write Faster, Better-Optimized JavaScript*. 2019. URL: https://medium.com/@bretcameron/13-tips-to-write-faster-better-optimized-javascript-dc1f9ab063d8 (besucht am 25. 01. 2021).

[can20a] canisuse.com. *WebAssembly Browser Support*. 2020. URL: https://caniuse.com/wasm (besucht am 12. 10. 2020).

[can20b] caniuse.com. *asm.js Browser Support*. 2020. URL: https://caniuse.com/asmjs (besucht am 19. 10. 2020).

[Chm+20] Jacek Chmiel u. a. *WebAssembly – Vierte Sprache des Web*. Avenga Germany GmbH. 2020. URL: https://www.avenga.com/de/magazine/webassembly-vierte-sprache-des-web/ (besucht am 29. 09. 2020).

[Chr19] Christophe. *Difference between direct and indirect function() calls*. 2019. URL: https://softwareengineering.stackexchange.com/questions/401110/difference-between-direct-and-indirect-function-calls (besucht am 10. 02. 2021).

[Chr20] Chromium projects and contributors. *Blink (Rendering Engine)*. 2020. URL: https://www.chromium.org/blink (besucht am 13. 10. 2020).

[Cla13] Clang Project. *Clang - Features and Goals*. 2013. URL: https://clang.llvm.org/features.html#gcccompat (besucht am 26. 10. 2020).

[Cla17a] Lin Clark. *A cartoon intro to WebAssembly*. Mozilla Foundation. 2017. URL: https://hacks.mozilla.org/2017/02/a-cartoon-intro-to-webassembly/ (besucht am 18. 09. 2020).

[Cla17b] Lin Clark. *A crash course in assembly*. Mozilla Foundation. 2017. URL: https://hacks.mozilla.org/2017/02/a-crash-course-in-assembly/ (besucht am 15. 10. 2020).

[Cla17c] Lin Clark. *A crash course in just-in-time (JIT) compilers*. Mozilla Foundation. 2017. URL: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/ (besucht am 02. 12. 2020).

[Cla17d] Lin Clark. *Memory in WebAssembly (and why it's safer than you think)*. Mozilla Foundation. 2017. URL: https://hacks.mozilla.org/2017/07/memory-in-webassembly-and-why-its-safer-than-you-think/ (besucht am 11. 02. 2021).

[Cla17e] Lin Clark. *WebAssembly table imports... what are they?* Mozilla Foundation. 2017. URL: https://hacks.mozilla.org/2017/07/webassembly-table-imports-what-are-they/ (besucht am 11. 02. 2021).

[Cla17f] Lin Clark. *What makes WebAssembly fast?* Mozilla Foundation. 2017. URL: https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/ (besucht am 19. 10. 2020).

[Cla18a] Lin Clark. *Calls between JavaScript and WebAssembly are finally fast.* 2018. URL: https://hacks.mozilla.org/2018/10/calls-between-javascript-and-webassembly-are-finally-fast-%F0%9F%8E%89/ (besucht am 05. 03. 2021).

[Cla18b] Lin Clark. *Making WebAssembly even faster: Firefox's new streaming and tiering compiler.* Mozilla Foundation. 2018. URL: https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler/ (besucht am 08. 12. 2020).

[Cla20a] Clang Project. *Clang: a C language family frontend for LLVM.* 2020. URL: https://clang.llvm.org/ (besucht am 08. 10. 2020).

[Cla20b] Clang Project. *Options That Control Optimization.* 2020. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html (besucht am 03. 12. 2020).

[Cla21] Clang Project. *Control Flow Integrity.* 2021. URL: https://clang.llvm.org/docs/ControlFlowIntegrity.html (besucht am 11. 02. 2021).

[Con21] CrypTool Contributors. *Cryptool Portal - Cryptography for everybody.* 2021. URL: https://www.cryptool.org/ (besucht am 08. 03. 2021).

[Ems20a] Emscripten Contributors. *Building to WebAssembly.* 2020. URL: https://emscripten.org/docs/compiling/WebAssembly.html (besucht am 27. 10. 2020).

[Ems20b] Emscripten Contributors. *Emscripten Compiler Frontend (emcc).* 2020. URL: https://emscripten.org/docs/tools_reference/emcc.html (besucht am 27. 10. 2020).

[Ems20c] Emscripten Contributors. *Emscripten Homepage.* 2020. URL: https://emscripten.org/ (besucht am 08. 10. 2020).

[Ems20d] Emscripten Contributors. *Emscripten SDK (emsdk).* 2020. URL: https://emscripten.org/docs/tools_reference/emsdk.html (besucht am 04. 11. 2020).

[Ems20e] Emscripten Contributors. *Interacting with code.* 2020. URL: https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html (besucht am 27. 10. 2020).

[Ems21] Emscripten Contributors. *Emscripten Runtime Environment.* 2021. URL: https://emscripten.org/docs/porting/emscripten-runtime-environment.html (besucht am 22. 01. 2021).

[Git20] GitHub. *WebAssembly Project on GitHub*. 2020. URL: https://github.com/WebAssembly (besucht am 13. 10. 2020).

[Goo19] Google LLC. *WebAssembly Migration Guide*. 2019. URL: https://developer.chrome.com/native-client/migration (besucht am 15. 10. 2020).

[Goo20a] Google LLC. *NaCl and PNaCl*. 2020. URL: https://developer.chrome.com/native-client/nacl-and-pnacl (besucht am 15. 10. 2020).

[Goo20b] Google LLC. *Welcome to Native Client*. 2020. URL: https://developer.chrome.com/native-client (besucht am 14. 10. 2020).

[Goo20c] Google LLC. *What is Native Client Good For?* 2020. URL: https://developer.chrome.com/native-client/faq#what-is-native-client-good-for (besucht am 14. 10. 2020).

[Haa+17] 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*. Hrsg. von Association for Computing Machinery. 2017, S. 185–200.

[HKZ20] Andreas Haas, Jakob Kummerow und Alon Zakai. *Up to 4GB of memory in WebAssembly*. Google LLC. 2020. URL: https://v8.dev/blog/4gb-wasm-memory (besucht am 31. 01. 2021).

[Jen20] Julian Jensen. *By when would WebAssembly have access to DOM?* 2020. URL: https://www.quora.com/By-when-would-WebAssembly-have-access-to-DOM (besucht am 02. 03. 2021).

[Kre18] Brian Krebs. *Who and What Is Coinhive?* 2018. URL: https://krebsonsecurity.com/2018/03/who-and-what-is-coinhive/ (besucht am 17. 02. 2021).

[LKP20] Daniel Lehmann, Johannes Kinder und Michael Pradel. „Everything Old is New Again: Binary Security of WebAssembly". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, S. 217–234.

[LLV19] LLVM team and contributors. *LLVM 8.0.0 Release Notes*. 2019. URL: https://releases.llvm.org/8.0.0/docs/ReleaseNotes.html (besucht am 26. 10. 2020).

[LLV20] LLVM team and contributors. *The LLVM Compiler Infrastructure*. 2020. URL: https://llvm.org/ (besucht am 25. 10. 2020).

[LLV21] LLVM team and contributors. *Getting Started with the LLVM System*. 2021. URL: https://llvm.org/docs/GettingStarted.html (besucht am 13. 01. 2021).

[McC17] Judy McConnel. *2017*. Mozilla Foundation. 2017. URL: https://blog.mozilla.org/blog/2017/11/13/webassembly-in-browsers/ (besucht am 05. 10. 2020).

[MDN20a] MDN contributors. *asm.js*. 2020. URL: https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js (besucht am 19. 10. 2020).

[MDN20b] MDN contributors. *Gecko*. Mozilla Foundation. 2020. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Gecko (besucht am 13. 10. 2020).

[MDN20c] MDN contributors. *Loading and running WebAssembly code*. Mozilla Foundation. 2020. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Loading_and_running (besucht am 12. 01. 2021).

[MDN20d] MDN contributors. *Memory Management*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management (besucht am 25. 01. 2021).

[MDN20e] MDN contributors. *WebAssembly Concepts*. Mozilla Foundation. 2020. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts (besucht am 19. 10. 2020).

[MDN21a] MDN contributors. *JavaScript typed arrays*. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays (besucht am 15. 01. 2021).

[MDN21b] MDN contributors. *WebAssembly.compile()*. Mozilla Foundation. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/compile (besucht am 29. 01. 2021).

[Mus+19a] Marius Musch u. a. „New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild“. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2019, S. 23–42.

[Mus+19b] Marius Musch u. a. „Thieves in the Browser: Web-Based Cryptojacking in the Wild“. In: *Proceedings of the 14th International Conference on Availability, Reliability and Security*. Association for Computing Machinery, 2019.

[Ook20] Ookla. *Speedtest Global Index*. 2020. URL: https://www.speedtest.net/global-index (besucht am 11. 12. 2020).

[RA18] Gedeon Rauch und Stephan Augsten. *Was ist Exception Handling?* 2018. URL: https://www.dev-insider.de/was-ist-exception-handling-a-775986/ (besucht am 26. 02. 2021).

[Rec20a] Sven Rech. *Msieve-Faktorisierung*. 2020. URL: https://www.cryptool.org/de/cto/msieve (besucht am 18. 03. 2021).

[Rec20b] Sven Rech. *Msieve-Web*. 2020. URL: https://github.com/ct-online/Msieve-Web (besucht am 18. 03. 2021).

[Rüt+18] Jan Rüth u. a. „Digging into Browser-Based Crypto Mining“. In: *Proceedings of the Internet Measurement Conference 2018*. Association for Computing Machinery, 2018, S. 70–76.

[Smi18] Ben Smith. *Don't allow IndexedDB serialization of WebAssembly.Module.* 2018. URL: https://github.com/WebAssembly/spec/issues/821 (besucht am 22. 02. 2021).

[Str11] J. Strombergson. *Test Vectors for the Stream Cipher RC4.* Internet Engineering Task Force (IETF). 2011. URL: https://tools.ietf.org/html/rfc6229 (besucht am 18. 12. 2020).

[Tec18] Technikbrennpunkt. *Coinhive – Krypto Mining statt Werbung? Was steckt dahinter, wie lohnenswert ist es und wie kann man es als Endnutzer verhindern.* 2018. URL: https://technikbrennpunkt.de/coinhive-krypto-mining-statt-werbung-was-steckt-dahinter-wie-lohnenswert-ist-es-und-wie-kann-man-es-als-endnutzer-verhindern-1101/ (besucht am 08. 03. 2021).

[Tho16] Seth Thompson. *Experimental support for WebAssembly in V8.* Google LLC. 2016. URL: https://v8.dev/blog/webassembly-experimental (besucht am 05. 10. 2020).

[UL20] UL. *PCMARK 10.* 2020. URL: https://benchmarks.ul.com/pcmark10 (besucht am 18. 12. 2020).

[Uni18] Marco Trivellato (Unity). *WebAssembly Load Times and Performance.* Unity Technologies. 2018. URL: https://blogs.unity3d.com/2018/09/17/webassembly-load-times-and-performance/ (besucht am 06. 01. 2021).

[W3C17a] World Wide Web Consortium (W3C). *Portability.* 2017. URL: https://webassembly.org/docs/portability/ (besucht am 07. 10. 2020).

[W3C17b] World Wide Web Consortium (W3C). *WebAssembly High-Level Goals.* 2017. URL: https://webassembly.org/docs/high-level-goals/ (besucht am 07. 10. 2020).

[W3C18a] World Wide Web Consortium (W3C). *Security.* 2018. URL: https://webassembly.org/docs/security/ (besucht am 25. 01. 2021).

[W3C18b] World Wide Web Consortium (W3C). *WebAssembly Core Specification.* 2018. URL: https://www.w3.org/TR/2018/WD-wasm-core-1-20180215/ (besucht am 05. 10. 2020).

[W3C18c] World Wide Web Consortium (W3C). *WebAssembly JavaScript Interface.* 2018. URL: https://www.w3.org/TR/2018/WD-wasm-js-api-1-20180215/ (besucht am 05. 10. 2020).

[W3C18d] World Wide Web Consortium (W3C). *WebAssembly Web API.* 2018. URL: https://www.w3.org/TR/2018/WD-wasm-web-api-1-20180215/ (besucht am 05. 10. 2020).

[W3C19] World Wide Web Consortium (W3C). *World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation*. 2019. URL: https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en (besucht am 17. 09. 2020).

[W3C20a] World Wide Web Consortium (W3C). *Design Rationale*. 2020. URL: https://github.com/WebAssembly/design/blob/master/Rationale.md (besucht am 04. 12. 2020).

[W3C20b] World Wide Web Consortium (W3C). *FAQ*. 2020. URL: https://webassembly.org/docs/faq/ (besucht am 19. 10. 2020).

[W3C20c] World Wide Web Consortium (W3C). *PARTICIPANTS IN THE WEBASSEMBLY COMMUNITY GROUP*. 2020. URL: https://www.w3.org/community/webassembly/participants (besucht am 05. 10. 2020).

[W3C20d] World Wide Web Consortium (W3C). *Roadmap*. 2020. URL: https://webassembly.org/roadmap/ (besucht am 13. 10. 2020).

[W3C20e] World Wide Web Consortium (W3C). *Use Cases*. 2020. URL: https://webassembly.org/docs/use-cases/ (besucht am 07. 10. 2020).

[W3C20f] World Wide Web Consortium (W3C). *WebAssembly Working Group - Participants*. 2020. URL: https://www.w3.org/groups/wg/wasm/participants (besucht am 06. 10. 2020).

[W3C20g] World Wide Web Consortium (W3C). *WebAssemlby Homepage*. 2020. URL: https://webassembly.org/ (besucht am 30. 09. 2020).

[Wag+17] Luke Wagner u. a. *WebAssembly consensus and end of Browser Preview*. World Wide Web Consortium (W3C). 2017. URL: https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html (besucht am 05. 10. 2020).

[Wag13] Luke Wagner. *asm.js in Firefox Nightly*. Mozilla Foundation. 2013. URL: https://blog.mozilla.org/luke/2013/03/21/asm-js-in-firefox-nightly/ (besucht am 19. 10. 2020).

[Wag15] Luke Wagner. *WebAssembly*. Mozilla Foundation. 2015. URL: https://blog.mozilla.org/luke/2015/06/17/webassembly/ (besucht am 05. 10. 2020).

[Wag16] Luke Wagner. *A WebAssembly Milestone: Experimental Support in Multiple Browsers*. Mozilla Foundation. 2016. URL: https://hacks.mozilla.org/2016/03/a-webassembly-milestone/ (besucht am 05. 10. 2020).

[Wik20a] Wikipedia. *Execution (computing)*. 2020. URL: https://en.wikipedia.org/wiki/Execution_(computing) (besucht am 14. 12. 2020).

[Wik20b] Wikipedia. *Runtime (program lifecycle phase)*. 2020. URL: https://en.wikipedia.org/wiki/Runtime_(program_lifecycle_phase) (besucht am 14. 12. 2020).

[Wik21] Wikipedia. *Control-flow integrity*. 2021. URL: https://en.wikipedia.org/wiki/Control-flow_integrity (besucht am 11. 03. 2021).

[Zak15] Alon Zakai. *EMSCRIPTEN AND WEBASSEMBLY*. Mozilla Foundation. 2015. URL: https://kripken.github.io/talks/wasm.html#/ (besucht am 27. 10. 2020).

[Zak16] Alon Zakai. *A WEBASSEMBLY TOOLCHAIN STORY*. Mozilla Foundation. 2016. URL: https://kripken.github.io/talks/emwasm.html#/ (besucht am 28. 10. 2020).

[Zak18] Alon Zakai. *Why do we need Binaryen-IR?* 2018. URL: https://github.com/WebAssembly/binaryen/issues/1520#issuecomment-385046238 (besucht am 28. 10. 2020).

[Zak19] Alon Zakai. *Emscripten and the LLVM WebAssembly backend*. Google LLC. 2019. URL: https://v8.dev/blog/emscripten-llvm-wasm (besucht am 27. 10. 2020).

[Zhu16] Limin Zhu. *Previewing WebAssembly experiments in Microsoft Edge*. Microsoft Corporation. 2016. URL: https://blogs.windows.com/msedgedev/2016/03/15/previewing-webassembly-experiments/ (besucht am 05. 10. 2020).

Abkürzungsverzeichnis

AOT Ahead-of-time
AST Abstract Syntax Tree
CFI Control Flow Integrity
CT3 CrypTool 3
CTO CrypTool-Online
DOM Document Object Model
emsdk Emscripten SDK
GCC GNU Compiler Collection
GUI Graphical User Interface
IETF Internet Engineering Task Force
IR Intermediate Representation
JS JavaScript
JSVM JavaScript Virtual Machine
LLVMIR LLVM Intermediate Representation
MDN Mozilla Developer Network
MVP Minimum Viable Product
NaCl Native Client
PNaCl Portable Native Client
SIMD Single Instruction Multiple Data
TTFB Time To First Byte
VM Virtuelle Maschine
W3C World Wide Web Consortium
Wasm WebAssembly

Abbildungsverzeichnis

1 WebAssembly-Logo 10
2 Meilensteine der WebAssembly-Entwicklung 11
3 Einordnung von Compiler-Front-, -Middle- und -Backend 14
4 Language Chain 15
5 Clang-LLVM-Toolchain 16
6 LLVM-Logo 17
7 Emscripten-Logo 18
8 Emscripten-Toolchain 18
9 Schritte, die Einfluss auf die Laufzeit von JS-Code haben 20
10 Schritte, die Einfluss auf die Laufzeit von Wasm-Code haben 21
11 Vergleich der Ausführungsschritte von JS und Wasm 21
12 Vergleich Parsen und Dekodieren 24
13 Gantt-Diagramm der JS-Ausführungszeit Teil 1: Parsen [Cla18b] 25
14 Gantt-Diagramm der Wasm-Ausführungszeit Teil 1: Parsen [Cla18b] . . . 25
15 Gantt-Diagramm der JS-Ausführungszeit Teil 2: Kompilieren [Cla18b] . . 27
16 Gantt-Diagramm der Wasm-Ausführungszeit Teil 2: Kompilieren [Cla18b] 28
17 Gantt-Diagramm der Wasm-Ausführungszeit Teil 3: Optimieren [Cla18b] 29
18 Leistung des Testsystems im Vergleich zu anderen Systemen 35
19 Vergleich der Gesamtgrößen der einzelnen Implementierungen 36
20 Vergleich der Ladezeiten der Web-Implementierungen 39
21 Vergleich der Ladezeiten des Unity Benchmark-Projekts 40
22 Vergleich der Laufzeiten 42
23 Logo des Mining-Services Coinhive 51
24 Screenshot einer in WebAssembly berechneten Animation 56
25 Notwendige Aufgaben vor der Nutzung von WebAssembly 60
26 Screenshot der Msieve-Faktorisierung 63
27 Stacktrace nach einer Exception in Wasm 64
28 ArrayBuffer mit mehreren Typed Array Views [MDN21a] 66
29 Optimierter JS-Code im Vergleich zu unoptimiertem JS-Code 68
30 Vergleich der Laufzeiten zwischen unterschiedlichen Desktop-Browsern . 69
31 Vergleich der Laufzeiten zwischen unterschiedlichen Smartphone-Browsern 70

Code-Listings

1 C-Code mit Buffer-Overflow-Schwachstelle 47
2 C-Code mit Schwachstelle zum Überschreiben eines Funktionspointers . . 48
3 Instantiierung über `WebAssembly.instantiate()` [MDN20c] 61
4 Instantiierung über `WebAssembly.instantiateStreaming()` [MDN20c] 61

Anhang

Auf der beiliegenden CD befindet sich die Masterarbeit als PDF-Dokument, der zugehörige LaTeX-Code und der Quellcode der verwendeten Beispiele.

Eidesstattliche Erklärung

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

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

(Datum) (Unterschrift)