



BACHELORARBEIT

Herr
Konrad Lux

**Vergleich der
Programmiersprache Rust mit C
und Java anhand einer
prototypischen Implementierung
eines Webserver**

2021

BACHELORARBEIT

Vergleich der Programmiersprache Rust mit C und Java anhand einer prototypischen Implementierung eines Webserver

Autor:

Konrad Lux

Studiengang:

Angewandte Informatik/IT-Sicherheit

Seminargruppe:

IF17wl-B

Matrikelnummer:

47434

Erstprüfer:

Prof. Dr.-Ing. Thomas Beierlein

Zweitprüfer:

Prof. Dr.-Ing. Uwe Schneider

Mittweida, Januar 2021

BACHELOR THESIS

Comparison of the programming language Rust with C and Java using a prototypical implementation of a webserver

Author:

Konrad Lux

Study Programme:

Applied computer science/IT security

Seminar Group:

IF17wl-B

Student number:

47434

First Referee:

Prof. Dr.-Ing. Thomas Beierlein

Second Referee:

Prof. Dr.-Ing. Uwe Schneider

Mittweida, January 2021

Bibliografische Angaben

Lux, Konrad: Vergleich der Programmiersprache Rust mit C und Java anhand einer prototypischen Implementierung eines Webservers, 91 Seiten, 16 Abbildungen, 31 Tabellen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Bachelorarbeit, 2021

Referat

Das Ziel dieser Arbeit ist es, die Möglichkeit der Nutzung von Rust als Ersatz für C und Java zu evaluieren. Diese beiden wurden als Vertreter von hardwarenahen bzw. einfacheren und höheren Sprachen gewählt. Sie sind außerdem bekannt dafür, unsicher bzw. langsam zu sein. In beiden Punkten soll Rust genauso gut oder besser sein als C und Java. Daher werden sie für die drei Sprachen verglichen. Da für die Wahl einer Programmiersprache auch entscheidend ist, wie aufwendig es ist, ein Programm zu erstellen, wird die Komplexität als dritter Aspekt für den Vergleich genutzt.

Anhand von Benchmarks wird zunächst die grobe Geschwindigkeit und Speichernutzung untersucht, während die Sicherheit der Sprachen in der Theorie und an einer realen Sicherheitslücke betrachtet wird.

Für einen ausführlichen Vergleich wird ein in allen drei Sprachen erstellter Webserver genutzt. Mittels diverser Metriken wird die Komplexität des Quellcodes analysiert und anschließend mit Laufzeitmessungen die Performanz.

Im Ergebnis ist Rust geeignet, C zu ersetzen, da es ähnlich schnell ist und ähnlich komplex, aber deutlich sicherer. Java hingegen ist zwar langsamer, aber dafür einfacher als Rust und sollte daher weiter eingesetzt werden für Programme, bei denen eine hohe Leistung nicht wichtig ist.

English title

Comparison of the programming language Rust with C and Java using a prototypical implementation of a webserver

Abstract

The goal of this thesis is to evaluate the possibility of Rust as replacement for C or Java. These two were selected as representative for low level/hardware-oriented and simple/high level programming languages, respectively. They are also known to be insecure and slow, respectively. Rust is supposed to be better than C and Java in both of these aspects. Therefore, they are compared for the three languages. As the effort to create a program is also important when choosing a programming language, the complexity will be the third aspect in this comparison.

At first, some benchmark programs will be used to get a rough estimate of the performance. The security of these languages will be reflected upon in theory and by means of a real vulnerability.

For a detailed comparison, a webserver created in each language will be used. Using different metrics, the complexity of the source code will be analysed, followed by a measurement of runtime performance.

It becomes clear that Rust is a valid replacement for C, as their performance and complexity are similar, but Rust is much more secure. Java on the other hand is slower, but much easier than Rust and should therefore be used for programs that don't need a high performance.

I. Inhaltsverzeichnis

Inhaltsverzeichnis	I
Abbildungsverzeichnis	II
Tabellenverzeichnis	III
Abkürzungsverzeichnis	IV
Danksagung	V
1 Einleitung	1
1.0.1 Gliederung	3
1.0.2 DVD	3
2 Programmiersprachen	4
2.1 Historie	4
2.2 C	6
2.2.1 Entstehung	6
2.2.2 Besonderheiten	7
2.3 Java	8
2.3.1 Entstehung	8
2.3.2 Besonderheiten	9
2.4 Rust	10
2.4.1 Grundlagen	10
2.4.2 Speicherverwaltung	12
2.4.3 Compiler	14
3 Vergleich von Rust mit C und Java	16
3.1 Komplexität	16
3.2 Performanz	18
3.2.1 Kompilierung	18
3.2.2 Laufzeit-Performanz	20
3.2.3 Speicherverbrauch	21
3.3 Sicherheit	23
3.3.1 Theorie	23
3.3.2 Beispiel einer Sicherheitslücke	32
4 Prototypischer Webserver als Vergleichsobjekt	34
4.1 Webserver	34
4.1.1 Historie	34
4.1.2 Technik	35
4.1.3 Eignung als Vergleichsobjekt	37
4.2 Anforderungen	38
4.2.1 Generelle Anforderungen	38

4.2.2	Grundlegende Funktionalität	38
4.2.3	HTTP	39
4.2.4	SSL/TLS	40
4.2.5	Fehlende Funktionen	40
5	Implementierungen des Webserver	42
5.1	Implementierung in C	42
5.1.1	Struktur	42
5.1.2	Ablauf	42
5.2	Implementierung in Java	45
5.2.1	Struktur	45
5.2.2	Ablauf	45
5.3	Implementierung in Rust	48
5.3.1	Struktur	48
5.3.2	Ablauf	48
5.3.3	Fremdbibliotheken	49
6	Vergleich der Implementierungen	51
6.1	Code-Komplexität	51
6.1.1	Codezeilen	52
6.1.2	Halstead-Metrik	52
6.1.3	Zyklomatische Komplexität	54
6.1.4	Kognitive Komplexität	54
6.1.5	gzip-Kompression	55
6.2	Performanz	56
6.2.1	Testsystem	57
6.2.2	Kompilierung	57
6.2.3	Laufzeit-Performanz	58
7	Fazit	63
7.1	Rust als Ersatz für hardwarenahe Sprachen wie C	63
7.2	Rust als Ersatz für Hochsprachen wie Java	64
	Literatur	65
A	Einführung Rust	71
A.1	Grundlagen	71
A.2	Besondere Strukturen	74
B	Tabellen	76
B.1	Kompilierzeiten Hello-World-Programme	76
B.2	Computer Language Benchmarks Game	78
B.3	Kompilierzeiten Webserver	83
B.3.1	C	83
B.3.2	Java	86
B.3.3	Rust	87

II. Abbildungsverzeichnis

3.1 Größe der Benchmark-Programme	17
3.2 Vergleich der Berechnungszeiten von Benchmarks für Programmiersprachen	20
3.3 Berechnungszeiten der Benchmarks	21
3.4 Speicherverbrauch der Benchmarks	22
3.5 Formatstring-Angriff	24
3.6 Pufferüberlauf	28
3.7 Beispiel einer <i>Race condition</i>	30
3.8 Speicherlayout bei <i>openssl-too-open</i>	33
4.1 Abruf einer Datei von einem Webserver	35
5.1 Dateien C-Server	44
5.2 Klassendiagramm Java	47
5.3 Moduldiagramm Rust	50
6.1 Kompilationszeiten der Webserver	58
6.2 Durchschnittliche Antwortzeit Webserver	59
6.3 CPU-Nutzung Webserver	60
6.4 Speichernutzung Webserver	62

III. Tabellenverzeichnis

3.1	Kompilierzeiten <i>Hello-World</i> -Programm	19
6.1	Anzahl Dateien und Zeilen Webserver	52
6.2	Halstead-Metrik Webserver	53
6.3	Zyklomatische Komplexität Webserver	54
6.4	Kognitive Komplexität Webserver	55
6.5	Größe der komprimierten Programme	56
6.6	Antwortdauer Webserver	59
6.7	Ressourcennutzung der Webserver	61
B.1	Kompilierzeiten eines <i>Hello-World</i> -Programmes in C	76
B.2	Kompilierzeiten eines <i>Hello-World</i> -Programmes in Java	76
B.3	Kompilierzeiten eines <i>Hello-World</i> -Programmes in Rust	77
B.4	Ergebnisse für <i>binary-trees</i> -Benchmark	78
B.5	Ergebnisse für <i>fannkuch-redux</i> -Benchmark	78
B.6	Ergebnisse für <i>fasta</i> -Benchmark	79
B.7	Ergebnisse für <i>k-nucleotide</i> -Benchmark	79
B.8	Ergebnisse für <i>mandelbrot</i> -Benchmark	80
B.9	Ergebnisse für <i>n-body</i> -Benchmark	80
B.10	Ergebnisse für <i>pidigits</i> -Benchmark	81
B.11	Ergebnisse für <i>regex-redux</i> -Benchmark	81
B.12	Ergebnisse für <i>reverse-complement</i> -Benchmark	82
B.13	Ergebnisse für <i>spectralnorm</i> -Benchmark	82
B.14	Kompilierzeit C-Webserver, kompiliert mit <i>gcc</i> (Debug ohne Optimierungen)	83
B.15	Kompilierzeit C-Webserver, kompiliert mit <i>gcc</i> (Debug mit Optimierungen (-Og)) . .	83
B.16	Kompilierzeit C-Webserver, kompiliert mit <i>gcc</i> (Release mit Optimierungen)	84
B.17	Kompilierzeit C-Webserver, kompiliert mit <i>clang</i> (Debug ohne Optimierungen) . . .	85
B.18	Kompilierzeit C-Webserver, kompiliert mit <i>clang</i> (Release mit Optimierungen) . . .	86
B.19	Kompilierzeit Java-Webserver	86

B.20	Kompilierzeit Rust-Webserver, kompiliert mit <i>rustc/cargo</i> (Debug ohne Optimierungen)	87
B.21	Kompilierzeit Rust-Webserver, kompiliert mit <i>rustc/cargo</i> (Release mit Optimierungen)	88
B.22	Kompilierzeit Rust-Webserver, kompiliert mit <i>rustc/cargo</i> (Debug ohne Optimierungen, inkl. Abhängigkeiten)	88
B.23	Kompilierzeit Rust-Webserver, kompiliert mit <i>rustc/cargo</i> (Release mit Optimierungen, inkl. Abhängigkeiten)	89

IV. Abkürzungsverzeichnis

CVE	Common Vulnerabilities and Exposures, Seite 32
GOT	Global Offset Table, Seite 32
HIR	High-Level Intermediate Representation, Seite 14
JNI	Java Native Interface, Seite 9
JVM	Java Virtual Machine, Seite 9
LLVM IR	LLVM Intermediate Representation, Seite 14
MIR	Mid-Level Intermediate Representation, Seite 14
RAII	Resource Acquisition Is Initialization, Seite 13
RCE	Remote Code Execution, Seite 7
RFC	Request for Comments, Seite 40
SLOC	Source Lines of Code, Seite 52
THIR	Typed High-Level Intermediate Representation, Seite 14

V. Danksagung

An dieser Stelle möchte ich mich bei denjenigen bedanken, die mir bei der Erstellung dieser Bachelorarbeit behilflich waren.

Zunächst geht mein Dank an Herrn Professor Beierlein und Herrn Professor Schneider von der Hochschule Mittweida, die sich bereit erklärt haben, diese Arbeit zu betreuen. Besonders bedanken möchte ich mich dabei bei Professor Beierlein, der mich während der Ausarbeitung mit Hinweisen und Korrekturen unterstützt hat.

Außerdem danke ich meiner Familie und meinen Freunden, die sich die Arbeit immer wieder durchgelesen haben und mich mit vereinten Kräften auf Fehler und unschöne Formulierungen hingewiesen und mir allgemein in vielerlei Hinsicht geholfen haben.

1 Einleitung

Es gibt viele Programmiersprachen. Wikipedia zählt fast 700 in der entsprechenden Liste [1] und beachtet dabei nur die wichtigen. Und immer noch werden es mehr. 30 der 700 sind seit 2010 erschienen.

Eine sticht dabei heraus.

Die Internetplattform `stackoverflow.com`, die viel von Softwareentwickler*innen genutzt wird, führt unter diesen seit 2011 jedes Jahr im Februar eine Umfrage durch [2]. Neben Fragen zur Person und dem beruflichen Umfeld wird auch ermittelt, welche Programmiersprachen sie besonders gerne verwenden. Aus der resultierenden Rangliste wird die *Most Loved Language* gekürt.

2020 gewann diesen Titel zum fünften Jahr in Folge die gleiche Sprache: Rust [3]. Bereits 2015 erreichte sie Platz 3 [4], obwohl es zu diesem Zeitpunkt noch nicht einmal eine stabile Version gab – Version 1.0 erschien erst im Mai.

Was ist Rust? C/C++, aber sicher. So lassen sich zumindest die Designziele kurz zusammenfassen. [5]

Viele gängige Programmiersprachen sind entweder sehr schnell (C/C++) oder einfach und sicher (Java, C#), aber nicht beides – heißt es zumindest. Das führt dazu, dass Programme, die hochperformant sein sollen, oft Sicherheitsprobleme haben. Für sichere Anwendungen muss hingegen auf eine andere Sprache ausgewichen werden und damit eine Einbuße bei der Leistung in Kauf genommen werden. Rust hingegen ist „rasend schnell“ ([6]) und garantiert trotzdem Speichersicherheit, sagen zumindest die Entwickler*innen.

Ein guter Kandidat also, um langsam, aber sicher C/C++-Programme zu ersetzen? Und Programme in Java und C# auch? Wenn man sich bisher zwischen schnell und sicher entscheiden musste und mit Rust nun nicht mehr, warum sollte man dann nicht alles ersetzen?

Dieses spannende Themengebiet ist Inhalt dieser Bachelorarbeit. Es wird der Frage nachgegangen, was denn tatsächlich die Unterschiede zwischen den drei Gruppen (schnell, sicher, Rust) sind. **Rust** wird dabei verglichen mit je einem Vertreter der beiden anderen Gruppen: mit **C** und **Java**.

Für eine Sprache ist allerdings nicht nur relevant, wie performant sie ist und wie sicher. Programme müssen geschrieben, verändert und verstanden werden können, und das von so vielen Menschen wie möglich. Eine Sprache, die zu kompliziert ist, um ein

Programm zu schreiben, ist sinnlos. Und: Wenn es in einer Sprache zwar möglich ist, ein sicheres Programm zu erstellen, es aber sehr viel einfacher ist, ein unsicheres zu programmieren, so wird ein großer Teil der Programme in dieser Sprache nicht sicher sein.

Damit ergeben sich drei relevante Aspekte, nach denen man die drei Programmiersprachen vergleichen kann: die **Sicherheit**, die **Performanz** und die **Komplexität**.

Struktur

Leider ist es nicht einfach, Programmiersprachen miteinander zu vergleichen. Eine Sprache kann nicht schneller sein als eine andere oder weniger Arbeitsspeicher verbrauchen; das können nur die Programme, die in den Sprachen geschrieben wurden.

Daher besteht diese Arbeit aus zwei Teilen: Im ersten wird versucht, aus allgemeinen Fakten und frei verfügbaren Informationen über die Sprachen eine Einschätzung bezüglich der drei oben genannten Aspekte zu erarbeiten.

Für den zweiten Teil wird in jeder Sprache ein Programm mit gleicher Funktionalität implementiert. Als Vergleichsobjekt wird ein prototypischer Webserver genutzt. Diese drei Programme werden anschließend stellvertretend für die drei Sprachen analysiert.

Manche der Eigenschaften lassen sich besser im allgemeinen, ersten Teil untersuchen, manche bezogen auf das Programm und im zweiten.

Es ist beispielsweise kaum möglich, die Performanz auf eine Sprache bezogen zu definieren. Dieses Kriterium bezieht sich stets auf ein Programm, und zwar in der Regel im Vergleich zu einem anderen Programm. Im ersten Teil werden daher für die Untersuchung dieses Aspekts mehrere kurze Beispielprogramme genutzt, während im zweiten Messungen an dem Webserver durchgeführt werden.

Die Komplexität ist ein subjektives Merkmal für Programmcode oder Programmiersprachen. Damit gibt es allerdings praktisch keine Möglichkeit, viele Sprachen in dieser Hinsicht zu vergleichen. Dafür wäre eine große Gruppe von Programmierer*innen nötig, die alle zu vergleichenden Sprachen auf einem ähnlichen Niveau beherrscht und darüber Auskunft gibt.

Stattdessen wird mittels objektiver Metriken versucht, diesen subjektiven Eindruck durch einen Zahlenwert auszudrücken. Durch diese Metriken lassen sich beliebige Programme in beliebigen Sprachen miteinander vergleichen – aber nicht die Sprachen selbst. Daher wird dieser Aspekt hauptsächlich im programmspezifischen zweiten Teil der Arbeit für die Webserver betrachtet.

Die Sicherheit eines einzelnen Programmes lässt sich hingegen praktisch nicht bestimmen. Stattdessen lässt sich nur aus der Anzahl der bisherigen Sicherheitslücken eine Abschätzung erstellen. Für ein frisch erstelltes Programm eignet sich diese Methode allerdings nicht. Dafür kann aus allgemeinen Informationen über eine Sprache eine Einschätzung erstellt werden. Diese Eigenschaft wird deshalb ausschließlich im ersten Teil erforscht.

1.0.1 Gliederung

Diese Arbeit besteht aus insgesamt sieben Kapiteln und zwei Anhängen.

Kapitel 2 beginnt mit einem kurzen Überblick über die Entwicklung von Programmiersprachen, um im Anschluss die drei untersuchten Sprachen genauer vorzustellen.

Ein erster allgemeiner Vergleich von ihnen erfolgt in Kapitel 3.

In Kapitel 4 wird erklärt, worum es sich bei einem Webserver überhaupt handelt und warum ein Webserver für den Vergleich gewählt wurde. Außerdem werden die Anforderungen an die zu programmierenden Webserver aufgelistet.

Im folgenden Kapitel 5 wird die Funktionalität der drei vollständigen Programme kurz erläutert.

Kapitel 6 vergleicht schließlich die Programme anhand verschiedener Aspekte.

Zuletzt wird in Kapitel 7 ein Fazit gezogen.

In Anhang A findet sich eine kurze Erläuterung der Rust-Syntax. Anhang B beinhaltet schließlich die Tabellen, die in den verschiedenen Abschnitten referenziert werden.

1.0.2 DVD

Auf der beiliegenden DVD befinden sich die erstellten Programme als Quellcode und ausführbare Binärdatei (alle für Linux und das Java- und Rustprogramm auch für Windows), und die generierte Dokumentation. Details zur Nutzung befinden sich in den README-Dateien.

Weiterhin sind hier die Python-Skripte, die für die Analyse in Kapitel 6 genutzt wurden, gespeichert.

Schließlich ist dort eine PDF-Datei dieser Arbeit zu finden.

2 Programmiersprachen

Ein Computer ist wenig hilfreich, solange er nichts macht. Daher muss ihm mitgeteilt werden, was er denn berechnen soll. Das wird ermöglicht durch sogenannte *Programmiersprachen*.

In diesem Kapitel wird zunächst die Entwicklung von Programmiersprachen betrachtet, um anschließend die drei untersuchten Sprachen genauer vorzustellen. Der Fokus liegt dabei auf *Rust*, da diese Sprache noch relativ jung und damit eher unbekannt ist.

2.1 Historie

Die Geschichte der Programmiersprachen beginnt fast zeitgleich mit der Entwicklung der ersten elektronischen Computer: Die erste Programmiersprache war der von Konrad Zuse in den 1940er-Jahren entwickelte *Plankalkül* [7, S. 10]. Die komplette Beschreibung wurde allerdings erst 1972 veröffentlicht [8, S. 41]; die erste Implementierung stammt von 1975 [9]. Praxisrelevanz erreichte die Sprache nie [7, S. 48].

Die Entwicklung der Programmiersprachen wird in mehrere Abschnitte, die sogenannten Generationen, eingeteilt. Die erste Generation waren Maschinensprachen, bei denen binäre, prozessorspezifische Befehle samt Daten an den Prozessor übergeben und von diesem direkt ausgeführt wurden [10, S. 1]. Ein Befehl verschob dabei beispielsweise Daten zwischen den Prozessorregistern oder führte arithmetische Operationen durch [7, S. 10].

In der folgenden Generation, den Assemblersprachen, wurden die Maschinencode-Befehle durch *Mnemonics* – möglichst kurze und verständliche Abkürzungen – ersetzt, was die entstehenden Programme lesbarer und einfacher zu verstehen machte [7, S. 11]. Ein Assembler übersetzte diesen Code dann in Maschinencode. Gleichzeitig ermöglichten *Binder* und Makroprozessoren einen modularen Aufbau des Programms aus mehreren Dateien bzw. die Erstellung von Funktionen, welche eine Menge von Befehlen zusammenfassten [10, S. 1–2].

Mit der dritten Generation erhöhte sich das Abstraktionslevel erheblich: So war es jetzt nicht mehr unbedingt notwendig, ein Programm speziell für einen bestimmten Prozessor zu erstellen [7, S. 11]. Dies war eines der Ziele, mit denen *ALGOL* geschaffen wurde [11, S. 27]; andere Programmiersprachen folgten. Damit wurde es möglich, Programme ohne genaue Kenntnis der Prozessorarchitektur zu schreiben. In *FORTRAN*, einer weiteren Sprache der dritten Generation, wurde die Ausdrucksabstraktion eingeführt, mit der beispielsweise eine Formel wie $A = B(I) + C$ in algebraischer Schreibweise dar-

gestellt werden konnte [10, S. 2]. Kontrollstrukturen wie Wiederholungen/Schleifen oder Bedingungen ersetzten nach und nach GOTO/JUMP-Anweisungen und sorgten so für einen übersichtlicheren Programmablauf [7, S. 11], [10, S. 4]. Zuerst in *ALGOL 60* enthalten war eine Bindung zwischen Daten und Datentypen [10, S. 2].

Neben den genannten ist *COBOL* ein weiterer wichtiger früher Vertreter dieser Generation [8, S. 6]. Diese Sprache wurde hauptsächlich für Geschäftsberichte geschaffen.

Neben diesen relativ universellen wurden auch Spezialsprachen entwickelt, die nur für einen bestimmten Anwendungszweck geeignet waren, diesen jedoch besonders einfach erfüllen konnten und so teilweise auch für Nichtinformatiker zu benutzen waren [10, S. 3], [7, S. 12]. Ein Beispiel dafür ist *GPSS* (General Purpose Simulation System), mit dem zeitlich diskrete Vorgänge simuliert werden können [10, S. 3].

Auch modernere Sprachen wie *C* [10, S. 7], Java oder auch Rust [12] werden der dritten Generation zugeordnet, da sich die grundlegende Programmstruktur nicht geändert hat.

Bei der vierten Generation steigert sich das Abstraktionslevel erneut. Programmierer*innen sollen nicht mehr den Algorithmus beschreiben, der das jeweilige Problem löst, sondern nur noch das Problem selbst [7, S. 12]. Dieses Prinzip wird auch *nicht-prozedural* genannt [10, S. 8]. Die Anweisungen ähneln dabei oft mehr der menschlichen Sprache als typischen Befehlen.

Als Beispiel sei SQL genannt, bei dem von den Programmierer*innen angegeben wird, *welche* Daten benötigt werden, und nicht, *wie* diese Daten aus einer Datenbank beschafft werden sollen [10, S. 8]. Ebenso zu dieser Generation zählen graphische Tools, die für den Nutzer unsichtbar im Hintergrund Code generieren [12].

Die Unterscheidung in Sprachen der dritten und vierten Generation ist nicht ganz eindeutig; viele Sprache besitzen Merkmale beider Generationen. Python hat beispielsweise eine Syntax, die in vielen Fällen an die englische Sprache angelehnt ist, würde ansonsten aber der dritten Generation zugeordnet.

Die fünfte Generation von Programmiersprachen weist keine fest definierten Merkmale wie die vorherigen Generationen auf. Je nach Quelle sind die Unterschiede zur vierten Generation anders festgelegt. Da jede Generation als Abstraktion der vorherigen Generation anzusehen ist und ein weiteres Abstraktionslevel nicht möglich sei, ist sogar die Existenz einer fünften Generation umstritten [13].

Auch in Sprachen dieser Generation geht es für Programmierer*innen eher darum, bestimmte Bedingungen festzulegen und daraus Schlüsse zu ziehen und nicht unbedingt selbst Code zu schreiben. Dies wird auch als *constraint programming* (etwa *Bedingungsprogrammierung*) oder *logic programming* (*Logikprogrammierung*) bezeichnet.

Als Beispiel für eine Sprache der fünften Generation wird *Prolog* (für **program**mation en **logi**que, Programmieren in Logik) angesehen [14]. In dieser Sprache werden Fakten in Form von logischen Aussagen und Relationen bzw. Regeln eingegeben und es können daraus logische Schlüsse gezogen werden.

2.2 C

Der TIOBE-Index bestimmt seit 2001 jeden Monat die Popularität einer Programmiersprache anhand der Anzahl der Websites, die sich damit beschäftigen. Und seit 2001 wurde der erste Platz jeden Monat von den gleichen zwei Sprachen eingenommen. Eine der beiden und die aktuelle Gewinnersprache ist C [15] – 50 Jahre nach Beginn der Entwicklung.

Woran liegt es, dass diese Sprache immer noch so populär ist? Und was ist bei der Programmierung in C zu beachten? Diese Fragen werden im Folgenden beantwortet.

2.2.1 Entstehung

Die Programmiersprache C wurde zu Beginn der 1970er-Jahre von Dennis Ritchie entwickelt und basiert unter anderem auf ALGOL 60, BCPL und B. Die beiden letzten Sprachen waren nicht typisiert, das heißt, jede Variable wurde als Maschinenwort gewertet, und nur durch den ausgeführten Befehl wurde klar, um was für einen Typ es sich handelt. Dieses Prinzip ist zwar einfach, führt aber zu einer Reihe von (Sicherheits-) Problemen, welche durch die Entwicklung von C eliminiert werden sollten. [16, S. 1], [8, S. 81, 82]

Eine weitere, durchaus wichtige Neuerung war der Präprozessor. Dieser wird vor dem Kompilervorgang ausgeführt und hauptsächlich genutzt, um Quellcode (in der Regel aus Header-Dateien) einzufügen, Makros zu ersetzen oder bestimmte Teile des Programms abhängig von Variablen zu entfernen (bedingte Übersetzung). Er wurde kurz nach Erstellung der Sprache eingeführt, wobei zu Beginn nur die Dateiinklusion und einfache Textersetzungen möglich waren. Zu diesem Zeitpunkt war der Präprozessor auch noch ein separates Programm (inzwischen ist er meist in Compiler integriert) und wurde nur aufgerufen, wenn die Quellcodedatei mit einem bestimmten Signal begann. [17, *Neonatal C*]

C führte nur wenige Features ein, die nicht schon in anderen Sprachen existierten. Die große Popularität verdankt diese Sprache stattdessen der Tatsache, dass ein kostenloser C-Compiler in UNIX – zu diesem Zeitpunkt ein viel benutztes Betriebssystem – integriert war. [8, S. 81, 82]

Die auch heute noch weite Verbreitung liegt auch daran, dass C eine vergleichsweise einfache Sprache mit wenigen Schlüsselwörtern ist, was den Compiler selbst und damit auch die Portierung auf neue Systeme einfacher macht als die von anderen Sprachen. [17, *Whence Success?*]

Obwohl der ursprüngliche Anwendungszweck die Systemprogrammierung war, ist C auch für eine Vielzahl anderer Programme geeignet ist. Die Standardbibliothek wurde

darauf angepasst, genau die Funktionen zu bieten, die von vielen Programmierer*innen regelmäßig verwendet wurden. [17, *Whence Success?*]

Zusammenfassend ist C also relativ einfach zu benutzen und gleichzeitig schnell und mächtig und füllte damit eine Lücke, die bis dahin keine andere Sprache belegt hatte.

2.2.2 Besonderheiten

Die Grundphilosophie von C ist, „dass die Programmierer wissen, was sie tun, und es nur explizit mitteilen müssen“ ([16, S. 3]). Es wird also davon ausgegangen, dass die Entwickler*innen keine Fehler machen und auch aus Compilersicht seltsame Anweisungen genau so ausgeführt werden sollen, wie sie programmiert sind. Dieses Prinzip ermöglicht es, sehr mächtige Programme zu schreiben. Allerdings sorgt es auch dafür, dass sehr schwerwiegende Probleme entstehen können, wenn Programmierer*innen etwas nicht genau oder falsch verstehen.

Für manche der Fehler gibt der Compiler (inzwischen) zumindest eine Warnung aus, wobei diese natürlich ignoriert werden können. Ein Beispiel dafür sind Formatstring-Angriffe (genauer beschrieben in Kapitel 3.3.1, Seite 23): Mit den richtigen Flags beim Kompilieren wird vom Compiler darauf hingewiesen, dass ein Sicherheitsproblem vorliegen könnte und ein solcher Angriff möglich ist. Standardmäßig sind diese Flags allerdings nicht aktiv.

Trotzdem gibt es noch genug Möglichkeiten, ein Programm zu schreiben, das augenscheinlich (richtig) funktioniert, aber in bestimmten Fällen fehlerhaft arbeitet. Ein Auslöser kann beispielsweise eine besondere Eingabe sein: Unerwartete oder einfach nur sehr viele Zeichen können unter Umständen den Ablauf des Programmes selbst manipulieren. Durch ein Speicherleck (*memory leak*) kann über einen langen Zeitraum der Arbeitsspeicher des ausführenden Computers belegt werden, bis es schließlich keinen freien Platz mehr gibt.

Die entstehenden Fehler sind vielfältig: Eine Rechnung könnte ein falsches Ergebnis liefern, oder das Programm könnte abstürzen. Kritischer sind vermutlich der Zugriff auf eigentlich nicht verfügbare Daten oder sogar die Möglichkeit, bei einem Angriff beliebigen Code auf dem System auszuführen (*Remote Code Execution*, RCE)). Genauer beschrieben wird dies in Kapitel 3.3 (Seite 23).

Ein großer Kritikpunkt ist die vergleichsweise schwache Typisierung in C [17], [18]. Variablen und Zeiger können – implizit oder explizit – in andere Typen umgewandelt werden, was zu Fehlern in Rechnungen oder Sicherheitslücken führen kann.

2.3 Java

In Kapitel 2.2 wurde erwähnt, dass die beiden Sprachen auf den Plätzen eins und zwei des TIOBE-Index seit 20 Jahren die gleichen sind. Um genau zu sein war C in diesem Zeitraum meist nicht die führende Sprache: Den ersten Platz belegte fast immer *Java* [15].

In dieser Sektion werden die Anfänge der Entwicklung dargelegt und die besonderen Merkmale aufgezeigt.

2.3.1 Entstehung

Java wurde ab 1991 von einem kleinen Team außerhalb der normalen Organisationsstruktur bei Sun Microsystems geschaffen. Das Ziel dabei war es, eine universelle und betriebssystemunabhängige Sprache zu entwickeln, welche die Kommunikation zwischen verschiedenen Geräten ermöglicht. Die entstehende Sprache, damals noch *Oak* genannt, war zunächst nur ein Beiprodukt für einen Prototyp eines PDA¹. [19], [20, S. 120–124]

Der Erfolg der Hardware blieb aus, die entwickelte Software sollte aber weiter verwendet werden. Daher wurde ein neuer Einsatzzweck gesucht und mit dem Internet auch gefunden. Ab 1994 war es die Bestimmung der Sprache, der Erschaffung interaktiver Anwendungen im Webbrowser zu dienen. Auch in diesem Jahr wurde sie aus markenrechtlichen Gründen in Java umbenannt. Unternehmensintern setzte sie sich gegen Mitbewerber als die Sprache durch, die in Zukunft von Sun genutzt und weiterentwickelt werden sollte. Damit sollte eine Teilung der Aufmerksamkeit und der Finanzen für die Entwicklung oder Unterstützung vieler verschiedener Sprachen verhindert werden. [20, S. 126–130]

Um die Sprache bekannt zu machen und eine breite Nutzerbasis zu erreichen, wurde sie als offener Standard mit genauen Spezifikationen, wie sie zu nutzen oder verändern ist, veröffentlicht. Dieses Vorgehen wurde von vielen Entwickler*innen geschätzt, die das dazu gegenteilige Verhalten von Microsoft oft kritisch sahen. Zusammen mit mehreren Artikeln in großen Zeitungen und der Zusage von Netscape, Java in den Netscape Navigator – damals der meistbenutzte Browser – zu integrieren, führte es dazu, dass Java 1995 in kurzer Zeit große Bekanntheit erlangte und mehrere Tausend Nutzer*innen erreichte [19]. Dies hatte auch den Vorteil für Sun, nicht in kurzer Zeit viele eigene Programmierer*innen ausbilden zu müssen. [20, S. 130–137]

Der breiten Öffentlichkeit bekannt wurde Java damals in der Form von sog. *Applets*, im Browser ausgeführten Java-Programmen. Im Quellcode der Seite wird dabei eine

¹ *Personal Digital Assistant*, ein kleiner, tragbarer Computer

Java-Klasse oder Datei referenziert, die heruntergeladen und lokal in der JVM (s. u.) ausgeführt wird [21, S. 68].

Diese Nutzung im Webbrowser existiert aktuell – aus Sicherheitsgründen und weil sie nicht auf Smartphones funktioniert – praktisch nicht mehr [22], [21, S. 68]. Dafür ist Java inzwischen eine vielbenutzte Programmiersprache, auch im Unternehmensumfeld. Die Einsatzzwecke reichen von Programmen für Endnutzer unter Android (Apps) oder Desktop-Betriebssystemen über serverseitige Anwendungen oder gar Webserver bis hin zu internen Programmen, beispielsweise für Big-Data-Analysen [23]. Aber auch das ursprüngliche Einsatzgebiet ist vertreten: Viele Blu-Ray-Player laufen mit Java [21, S. 47].

2.3.2 Besonderheiten

Bald nach der Veröffentlichung wurde die *Java Virtual Machine* (JVM) eingeführt, die als virtueller Computer Java-Programme auf jeder beliebigen Plattform ausführen kann [20, S. 142]. Der Programmcode wird dabei in sog. *Bytecode* übersetzt, der von dem (plattformspezifischen) virtuellen Prozessor während der Laufzeit kompiliert, optimiert und ausgeführt wird [21, S. 50, 53–54]. Von Programmierer*innen werden die von der JVM bereitgestellten, plattformunabhängigen Funktionen genutzt, welche die jeweiligen Betriebssystemfunktionen aufrufen [21, S. 51]. Die JVM bietet auch einige Zusatzfunktionen wie einen *Garbage Collector*, der für die automatische Speicherbereinigung zuständig ist, eine starke Typprüfung und ein klar definiertes Speicher- und Threading-Modell. [21, S. 50]

Die Fehlerbehandlung erfolgt in Java über *Ausnahmen* (*Exceptions*). Diese können von Funktionen als Reaktion auf bestimmte Ereignisse geworfen (*throw*) werden. Eine andere Funktion im Aufrufstack kann diese Ausnahme abfangen (*catch*) und behandeln. Falls dies nicht passiert, wird das Programm beendet und die Ausnahme inklusive Aufrufstack als Fehlermeldung ausgegeben. Ein Beispiel dafür ist der Zugriff auf ein Array-Element außerhalb der Grenzen, was zu einer *Exception* führt. Das Lesen oder Schreiben über Array-Grenzen hinweg – in C/C++ ein großes Sicherheitsproblem – kommt so mit Java nicht vor. [21, S. 58, 62]

Direkt auf die Hardware eines Systems oder auf bestimmte Speicherzellen zuzugreifen, ist in Java nicht möglich. Damit ist Java für die Systemprogrammierung und Treiberentwicklung nicht geeignet. Ist dies gefordert, müssen über das *Java Native Interface* (JNI) die jeweiligen Systemfunktionen aufgerufen werden. Diese müssen allerdings in einer nativen Programmbibliothek für jede Plattform einzeln verfügbar gemacht werden. [21, S. 62–63]

2.4 Rust

Die Entwicklung von Rust begann 2006 als privates Projekt des Mozilla-Mitarbeiters Graydon Hoare; die Finanzierung durch Mozilla startete 2009 [24]. Offiziell vorgestellt wurde die Sprache auf dem *Mozilla Summit 2010*. Seit 2011 ist der in Rust geschriebene Compiler in der Lage, eine neue Version von sich selbst zu übersetzen [25].

Ab 2012 wurde Rust genutzt und entwickelt, um eine sicherere und trotzdem schnelle Browser-Engine (*Servo*) zu erschaffen [26]. Teile dieser Engine wurden seitdem in die aktuell genutzte Browser-Engine *Gecko* übernommen; seit 2017 wird für jeden Bau von Firefox der Rust-Compiler benötigt [27].

2015 wurde Version 1.0 und damit die künftige Verpflichtung zur Abwärtskompatibilität veröffentlicht [28].

2.4.1 Grundlagen

Die Grundprinzipien, auf denen Rust basiert, lauten [29]:

Praxisnähe Rust soll eine Sprache sein, die für Probleme im praktischen Einsatz genutzt werden kann.

Pragmatismus Es sollen Zugeständnisse für die Benutzerfreundlichkeit und die Integration in heutige Systeme zugelassen werden.

Speichersicherheit Die Sprache muss Speichersicherheit erzwingen und darf keine Segmentierungsfehler und andere Speicherzugriffsverletzungen erlauben.

Leistung Sie muss in der gleichen Leistungsklasse wie C++ sein.

Nebenläufigkeit Sie muss moderne Lösungen für das Schreiben von nebenläufigem Code bieten.

Es ist außerdem erklärtes Ziel, möglichst kompatibel zu C zu sein [5]. Über sogenannte *Foreign Function Interfaces* (FFI) auf Nutzerseite und *Application Binary Interfaces* (ABI) auf Geberseite ist es möglich, Funktionen in anderen Sprachen (wie C) aufzurufen oder von anderen Sprachen (wie C) aus Rust-Funktionen zu starten [30, Kap. 19.1].

Programmstruktur

Ein Rust-Programm besteht aus einem *package*, welches mindestens eine *crate* enthält. Eine *crate* ist unterteilt in Module (*modules*), welche selbst mehrere Module beinhalten können. [30, Kap. 7.1]

Innerhalb eines Modules kann beliebig auf alle Elemente in Rust (Funktionen, Methoden, *structs*, *enums*, Module und Konstanten) zugegriffen werden. Falls Elemente ande-

rer Module genutzt werden sollen, müssen diese (mit `pub`) explizit zugänglich gemacht werden; standardmäßig sind sie nur innerhalb des Moduls nutzbar.

Generell muss für die Nutzung eines anderen Moduls stets der Pfad dahin angegeben werden. Über das `use`-Schlüsselwort können aber auch Elemente in den eigenen Geltungsbereich inkludiert werden, wodurch der Pfad zum jeweiligen Element nicht mehr gebraucht wird. [30, Kap. 7.3]

Fremdbibliotheken

Die Rust-Community stellt unter <https://crates.io/> ein Paketarchiv bereit, zu dem Programme und Bibliotheken in Form von *crates* hochgeladen werden können. Diese *crates* lassen sich für eigene Programme nutzen, indem der Name und die Version in eine Manifest-Datei eingetragen werden. Beim Bau des Programms bzw. der Bibliothek werden durch den Rust-Paketmanager *cargo* alle diese Abhängigkeiten inklusive deren Abhängigkeiten vom Server geladen und kompiliert. [31, Kap. 2.4]

Unveränderbarkeit von Variablen

Variablen und Referenzen sind in Rust standardmäßig nicht veränderbar (*immutable*). Eine zweite Zuweisung zu einer Variablen oder eine Änderung führen zu einem Fehler beim Kompilieren. Falls das nötig ist, muss die Veränderbarkeit (*mutability*) mit dem Schlüsselwort `mut` explizit festgelegt werden. [30, Kap. 3.1]

Typisierung

Rust ist stark typisiert: Ein Element kann seinen Typ nicht ändern, implizite Typumwandlungen gibt es nicht [32, Kap. 5.1]. Im Kompilierprozess wird eine statische Typprüfung durchgeführt, wodurch Fehler nicht erst zur Laufzeit auffallen. Es ist allerdings normalerweise nicht notwendig, einen Typ anzugeben. Stattdessen kann der Compiler diesen ableiten, selbst wenn bei der Variablendeklaration kein Initialwert festgelegt wird. Falls allerdings ein Programmpfad keinen Wert zuweist, führt das zu einem Fehler beim Kompilieren. [30, Kap. 3.2]

Makros

In vielen Fällen kommen in Rust *Makros* zum Einsatz. Makros lassen sich als eine verbesserte Art von Präprozessor-Direktiven beschreiben: Sie werden ebenfalls beim Kompilieren aufgelöst, allerdings wird keine reine Textersetzung vorgenommen, son-

dern es werden auch Datentypen und der Kontext beachtet. [30, Kap. 19.5] erklärt ihre Funktionsweise als *Metaprogrammierung*: Code, der neuen Code erzeugt.

Es wird unterschieden zwischen *deklarativen* und *prozeduralen Makros*. *Deklarative Makros* vergleichen, ähnlich wie *switch*-Anweisungen, eine Variable – in diesem Fall einen Programmcode-Abschnitt – mit bestimmten Mustern und führen bei einer Übereinstimmung eine Ersetzung durch. *Prozedurale Makros* sind dagegen Rust-Funktionen, die Quellcode (in Form eines *TokenStreams*) als Eingabe übergeben bekommen und daraus neuen Rust-Code erzeugen. [30, Kap. 19.5]

unsafe

Funktionen und Blöcke können mit dem Schlüsselwort `unsafe` als unsicher deklariert werden. Es kann in bestimmten Fällen nötig sein, Prüfungen des Compilers zu umgehen: „Sometimes, however, it is necessary to write code that is correct for reasons which are too clever for the compiler to understand. In those cases, you need to use Unsafe Rust.“ Die Korrektheit des Programmes sicherzustellen liegt dann in der Verantwortung der Programmierer*innen. [33, Keyword *unsafe*]

Diese Funktionalität sollte jedoch sparsam und nur in speziellen Fällen genutzt werden. Über das Schlüsselwort wird auf die Verpflichtung zu erhöhter Aufmerksamkeit hingewiesen, schließlich werden viele Tests des Rust-Compilers mit der Verwendung von *unsafe* deaktiviert.

Der Zugriff auf in anderen Sprachen geschriebene Funktionen oder von diesen auf Rust-Funktionen ist stets *unsafe*, da der Rust-Compiler keine Möglichkeit hat, in externem Code Prüfungen durchzuführen.

2.4.2 Speicherverwaltung

Eines der Hauptmerkmale von Rust ist das *Ownership*-Prinzip, das das Verhältnis einer Variablen zu ihrem Wert erklärt. Zusammengefasst wird es in [30, Kap. 4.1] mit drei Regeln:

- Jeder Wert in Rust gehört einer Variablen, die seine *Eigentümerin* (*owner*) genannt wird.
- Es kann zu jedem Zeitpunkt immer nur eine Eigentümerin geben.
- Wenn der Eigentümerin den aktuellen Gültigkeitsbereich (*Scope*) verlässt, wird der Wert verworfen.

Die dritte Regel ist typisch für das Prinzip *Resource acquisition is initialization* (RAII), welches beispielsweise auch in C++ zu finden ist. Sie besagt, dass beim Verlassen des Gültigkeitsbereichs der Destruktor für alle lokalen Variablen aufgerufen und der Speicherplatz freigegeben wird.

Würden allerdings mehrere Variablen auf den gleichen Speicherbereich verweisen, würden deren Destrukturen diesen wiederholt freigeben. Dieses Verhalten wird *double free* genannt und ist ein typisches Problem in C (vgl. Kapitel 3.3.1, Seite 31). Um dies zu verhindern, gibt es die zweite Regel, laut der es nicht möglich ist, den Wert einer Variablen zu kopieren. Stattdessen wird bei einer entsprechenden Anweisung der Wert verschoben (*moved*). Ein Zugriff auf die erste Variable ist dann nicht mehr erlaubt und führt zu einem Fehler beim Kompilieren. Davon ausgenommen sind Variablen, deren Wert zur Kompilierzeit feststeht und auf dem Stack gespeichert ist, also beispielsweise Ganzzahlen, Booleans oder Zeichen (genauer: alle Typen, die den *Copy*-Trait besitzen. *Traits* legen, ähnlich wie *Interfaces* in anderen Sprachen, eine Reihe von Methoden fest, die ein Typ implementieren muss).

Falls dieses Verschieben nicht gewünscht ist, muss explizit eine Kopie (*clone*) angefertigt werden. Dabei wird der gesamte Speicher kopiert, was unter Umständen lange dauert.

Beim Aufruf einer Funktion werden die übergebenen Argumente kopiert, wenn sie den *Copy*-Trait besitzen, und ansonsten verschoben. Im zweiten Fall sind sie damit normalerweise nach dem Funktionsaufruf nicht mehr nutzbar. Ansonsten müssen sie von der Funktion zurückgegeben werden – auch dabei werden sie verschoben. [30, Kap. 4.1]

Eine Alternative ist es, nicht die Variable selbst als Argument zu übergeben, sondern nur eine Referenz. Das ist auch sonst möglich: Es können beliebig viele (unveränderbare) Referenzen auf eine Variable erstellt werden, ohne dass sich die Eigentümerin ändert. In Rust heißt dieser Vorgang Leihen (*borrow*).

Auch Referenzen können veränderbar sein und sind es aber standardmäßig nicht. Mit einer einfachen Referenz lässt sich also das referenzierte Element nicht verändern.

Es kann allerdings zu jedem Zeitpunkt nur eine veränderbare Referenz (*mutable borrow*) existieren und auch dies nicht gleichzeitig mit unveränderbaren Referenzen.

Auf diese Weise sollen *data races* verhindert werden. *Data races* sind Situationen, in denen gleichzeitig von mehreren Ausführungssträngen auf einen Speicherbereich zugegriffen und er von mindestens einem verändert wird (vgl. Kapitel 3.3.1, Seite 29). Dadurch kann der gelesene oder geschriebene Wert inkorrekt sein.

Ein häufiges Problem bei der Arbeit mit Referenzen sind sog. *dangling pointers* oder *dangling references*: Referenzen, die auf einen Speicherbereich verweisen, der bereits freigegeben wurde (vgl. Kapitel 3.3.1, Seite 31). Dies kann beispielsweise passieren, wenn aus einer Funktion eine Referenz auf eine lokale Variable zurückgegeben wird.

Beim Verlassen der Funktion wird der Speicherplatz der Variablen freigegeben, die Referenz allerdings nicht verändert. Eine weitere Nutzung der Referenz ist undefiniertes Verhalten und kann zu Fehlern und Sicherheitslücken führen.

Rust verhindert dieses Problem, indem der Compiler prüft, ob Rückgabewerte eine Referenz auf lokale Variablen enthalten und in diesem Fall eine Fehlermeldung ausgibt. Damit ist für Programmierer*innen sichergestellt, dass jede Referenz zu jedem Zeitpunkt gültig ist. [30, Kap. 4.2]

2.4.3 Compiler

Der Kompilationsprozess besteht bei Rust aus zwei Schritten: Zuerst wird der Rust-Code vom Rust-Compiler (*rustc*) in einen Zwischencode übersetzt, die sogenannte LLVM Intermediate Representation (*LLVM IR*). Dieser Code wird im Anschluss vom LLVM-Compiler in Maschinencode kompiliert.

Der Rust-Compiler bildet dabei das sog. *Compiler-Frontend*, welches für die Analyse des Quellcodes zuständig ist. Dieser wird beim Aufruf zunächst mittels eines *Tokenizers* bzw. *Lexers* in *Tokens* (kleine, unteilbare Abschnitte) aufgeteilt. Diese Tokens werden validiert und Strings interniert (darunter ist die Ersetzung von gleichen Strings durch Referenzen auf eine einzige Instanz zu verstehen). Anschließend wird aus den Tokens ein abstrakter Syntaxbaum (*abstract syntax tree*) geparkt, welcher in eine *High-Level Intermediate Representation* (HIR) umgewandelt wird. In dieser Stufe findet die *type inference* statt, also die Erkennung des Typs eines Ausdrucks.

Nun wird die HIR zunächst in eine *Typed High-Level Intermediate Representation* (THIR) konvertiert, auf welcher die Validierung von Mustern für *match*-Ausdrücke (Mehrfachverzweigung, ähnlich zu *switch*) durchgeführt wird, und daraus in eine *Mid-Level Intermediate Representation* (MIR).

Diese Repräsentation wird für *borrow checks* genutzt. Diese bestehen aus einer Reihe von Prüfungen: Wurden alle Variablen vor ihrer Nutzung initialisiert? Wurde ein Wert mehrfach verschoben? Wurde ein Wert verschoben, während eine Referenz auf ihn existiert? Gibt es eine Zugriffsmöglichkeit auf Speicherbereiche, für die eine veränderbare Referenz existiert (außer dieser Referenz)? Kann ein Wert verändert werden, während eine nichtveränderbare Referenz existiert? [34, Kap. 43] Außerdem werden erste Optimierungen am Programm durchgeführt.

Schließlich wird die MIR in die *LLVM Intermediate Representation* (LLVM IR) umgewandelt.

Generics werden dabei in die jeweils benutzten Varianten aufgelöst (*monomorphized*) – ähnlich zu *Templates* in C++. Der Begriff *generic* bezeichnet die Nutzung abstrakter Typen, welche später mit einem konkreten Typ aufgerufen werden. Ein Beispiel ist eine generische Liste für Objekte eines Typs, die als Liste von Ganzzahlen oder als Liste von Booleans genutzt werden kann. Der Compiler generiert daraus einen Typ für eine

Ganzzahlenliste und einen Typ für eine Liste von Booleans; die generische Liste verschwindet.

Damit wird das *zero-overhead*-Prinzip erfüllt, welches besagt: Was nicht benutzt wird, verbraucht keine Ressourcen, aber was benutzt wird, könnte auch manuell nicht besser geschrieben werden. Der Polymorphismus wird also aufgelöst und ist im Maschinencode nicht mehr zu sehen; es entsteht auch kein Overhead durch weitere mögliche Typen. Im Programm wurden aber generische Funktionen verwendet, anstatt die gleiche Funktion mehrfach für alle verwendeten Typen zu schreiben. Dieser Ansatz sorgt zwar dafür, dass sich das übersetzte Programm vergrößert, weil genau die verwendeten Funktionen für alle Typen vorhanden sind anstatt einer generischen. Durch die Typprüfungen, die so entfallen können, erhöht sich dennoch die Performanz. [35]

Die LLVM IR wird anschließend an den LLVM-Compiler übergeben und dort weiter optimiert. Damit beginnt die Synthesephase im sog. *Backend* des Compilers. Die Ausgabe ist der Maschinencode für die jeweilige Plattform. Zum Abschluss werden noch alle genutzten *Binaries* und Bibliotheken zur endgültigen Binärdatei gelinkt. [34, Kap. 20]

3 Vergleich von Rust mit C und Java

In diesem Kapitel findet ein erster Vergleich der drei Programmiersprachen statt. Dabei werden Benchmarks und Testprogramme zur Messung der Komplexität und Performanz genutzt, während die Sicherheit theoretisch und anhand einer realen Sicherheitslücke betrachtet wird.

3.1 Komplexität

Das *Computer Language Benchmarks Game* [36] vergleicht regelmäßig anhand einer Reihe von *Benchmark-Programmen*² unter anderem die Geschwindigkeit verschiedener Programmiersprachen. Diese Benchmarks sind Programme, die hauptsächlich mathematische Berechnungen in allen ausgewählten Programmiersprachen durchführen. Insgesamt werden für jede Sprache zehn Benchmarks bearbeitet: *binary-trees*³, *fannkuch-redux*, *fasta*, *k-nucleotide*, *mandelbrot*, *n-body*, *pidigits*, *regex-redux*, *reverse-complement* und *spectralnorm*. Dabei wurden die Benchmarks in jeder Sprache von verschiedenen Programmierer*innen mehrfach implementiert. Damit sollen Fehler der Implementierung, die zu schlechten Werten führen, vermindert oder aufgeschlossen werden. Im Optimalfall soll so eine für die Sprache perfekte Implementierung des jeweiligen Algorithmus erreicht werden, um damit die Sprache selbst bewerten oder vergleichen zu können. Die Programme in verschiedenen Sprachen werden dabei als untereinander gleich angenommen.

Unter diesen Voraussetzungen lässt sich aus der Komplexität der Programme ein Maß für die Komplexität der Programmiersprache ableiten. Um erstere zu bestimmen, wird die Länge der Programme bewertet. Dafür werden zuerst Kommentare und unnötige *Whitespace*-Zeichen (Leerzeichen, Tabulatoren, Zeilenumbrüche, etc.) entfernt. Anschließend wird der so gereinigte Code mit *gzip* komprimiert.

In Abbildung 3.1 (Seite 17) ist das Ergebnis für die jeweils schnellste Version einer Programmiersprache pro Benchmark dargestellt. Das komprimierte C-Programm ist in sieben der zehn Fälle das kleinste, bei dem *mandelbrot*-Benchmark dafür das größte. Rust stellt dreimal den zweitbesten Wert, Java fünfmal. Dafür ist die Rust-Implementierung von *binary-trees* und *mandelbrot* die kürzeste, Java schafft dies nur einmal für *regex*-

² *Benchmark-Programme* oder kurz *Benchmarks* sind Programme, die eine bestimmte Aufgabe mehrfach unter verschiedenen Bedingungen (andere Prozessoren, Betriebssysteme, Compiler, etc.) ausführen und damit eine Vergleichbarkeit erschaffen

³ Eine Beschreibung der einzelnen Benchmarks ist auf der jeweiligen Seite zu finden. Diese sind verlinkt unter <https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/binarytrees.html>

redux. Insgesamt liegt C hier also vorne, während zwischen Java und Rust keine klare Abstufung möglich ist.

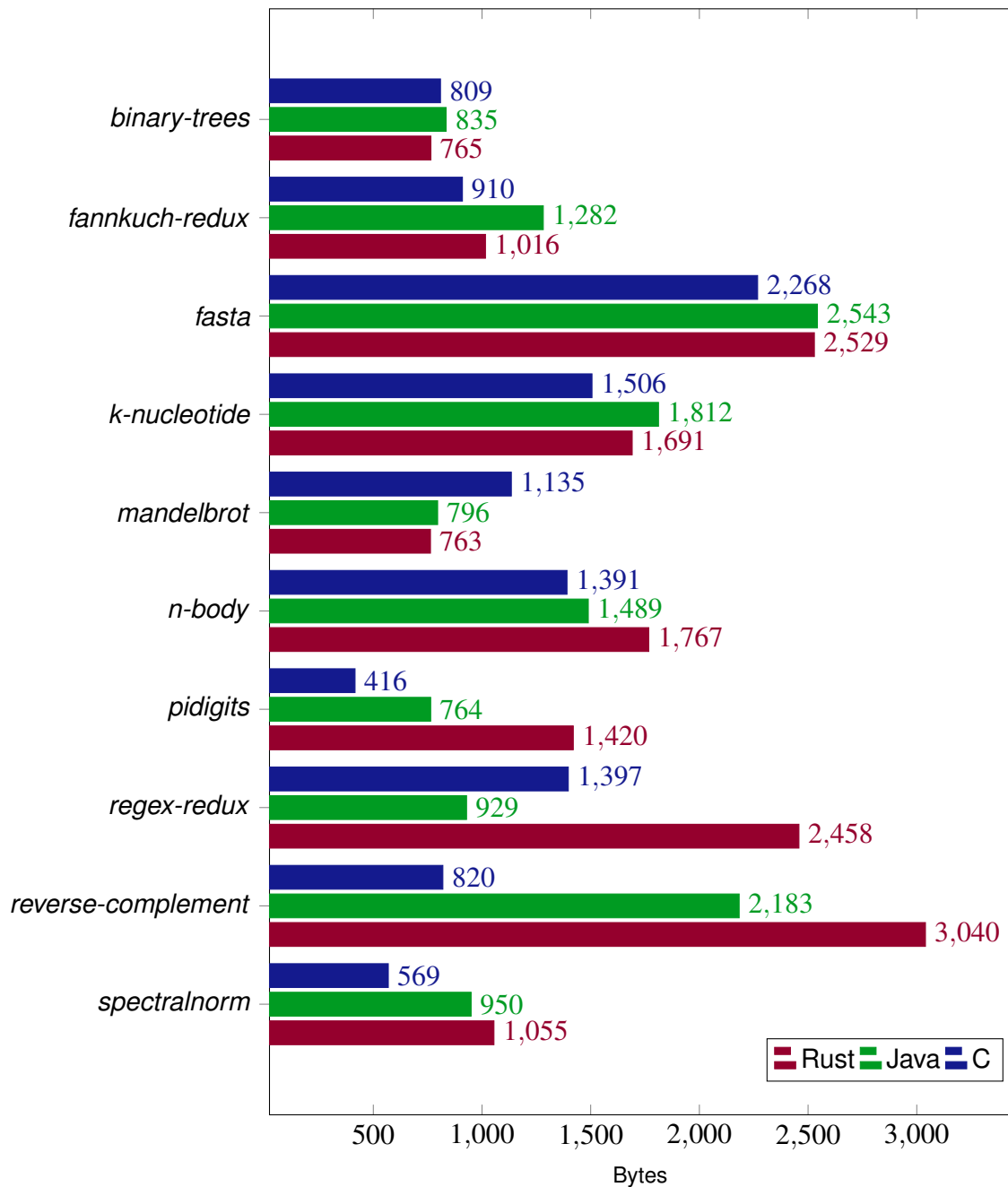


Abbildung 3.1: Größe der Benchmark-Programme pro Algorithmus (bereinigt, komprimiert mit *gzip*): schnellste Zeit aus Tabellen B.4 bis B.13 (S. 78 ff.).

Quelle: [37]

3.2 Performanz

Die Performanz eines Programmes setzt sich zusammen aus der Zeit, die für eine bestimmte Arbeit benötigt wird, und dem dabei genutzten Arbeitsspeicher. Beide sollten so niedrig wie möglich sein. Zusätzlich wird der Kompiliervorgang betrachtet.

3.2.1 Kompilierung

Der Compiler von Rust ist hauptsächlich auf eine niedrige Laufzeit (*runtime*) ausgelegt, die Kompilierzeit (darin ist in diesem Abschnitt das *Linken* mit eingeschlossen) ist dafür erwähnenswert hoch. Ein einfaches Beispiel ist die Kompilierzeit für ein *Hello-World*-Programm:

```
----- Hello-World-Programm in C -----  
1 #include <stdio.h>  
2  
3 int main() {  
4     printf("hello, world\n");  
5     return 0;  
6 }
```

```
----- Hello-World-Programm in Java -----  
1 public class HelloWorld {  
2     public static void main(String[] args) {  
3         System.out.println("hello, world");  
4     }  
5 }
```

```
----- Hello-World-Programm in Rust -----  
1 fn main() {  
2     println!("hello, world");  
3 }
```

Wie in Tabelle 3.1 zu sehen ist, dauert der Kompiliervorgang für Rust durch *rustc* mit 226 ms im Durchschnitt fünfmal so lange wie mit C und *gcc* (45 ms). Java liegt mit

498 ms an letzter Stelle, wobei dies vermutlich auf Overhead beim Starten des Compilers zurückzuführen ist.

Die für die Tests genutzte Hard- und Software ist beschrieben in Kapitel 6.2.1 auf Seite 57.

Tabelle 3.1: Durchschnittliche Kompilierzeit von *Hello-World*-Programmen in Sekunden, aufgeteilt in Gesamtzeit und Rechenzeit im Nutzer- und Systemkontext
Durchschnittswerte aus Tabellen B.1 bis B.3, Seite 76 f.

Sprache	real	user	sys
C	0,045	0,031	0,014
Java	0,498	1,450	0,080
Rust	0,226	0,192	0,065

Von den Nutzer*innen von Rust werden die langen Kompilierzeiten bemängelt; eine Verbesserung dieses Vorgangs ist ein Wunsch von vielen von ihnen [38].

Von den Rust-Entwickler*innen werden mehrere Erklärungen dafür genannt. Der erste ist ein anderes Kompiliermodell: Eine Kompilereinheit in C/C++ ist eine Datei, in Rust aber eine sog. *crate*, welche aus vielen Dateien bestehen kann. Ändert sich eine Datei davon, muss die gesamte *crate* neu übersetzt werden.

Das komplexere Typsystem und die entsprechenden Prüfungen sind ein weiterer Grund, genau wie die Analyse beim Leihen (*borrowing*) von Zeigern. Schließlich ist auch die Nutzung von LLVM als Teil des Kompilierens ein Problem: LLVM „erzeugt guten Maschinencode, läuft aber relativ langsam“ ([29]). Hierbei ist es ebenfalls nicht hilfreich, dass der Rust-Compiler aktuell in Teilen technisch mangelhaft ist („the Rust compiler suffers from long-standing technical debt“ ([39])) und daher die Qualität des generierten und von LLVM verarbeiteten Codes (*LLVM Intermediate Representation*) schlecht ist. [39]

Dieses Problem wurde während des Schaffungsprozesses von Rust zunächst zugunsten von höherer Laufzeit-Performanz ignoriert, aber seit 2015 mit Version 1.0 verstärkt bearbeitet. Seitdem wurde mit einer Reihe von Maßnahmen und Optimierungen versucht, entgegenzuwirken. Beispiele sind (bessere) Parallelisierung des Rust-Compilers und Optimierungen bei der Erzeugung von LLVM-Code, aber auch verschiedene Analysemöglichkeiten. [29]

Inzwischen sind erste Fortschritte zu erkennen: Laut [40] findet die Hälfte der befragten Entwickler*innen, dass sich die Kompilierzeiten verbessert haben.

3.2.2 Laufzeit-Performanz

Die Seite *Computer Language Benchmarks Game* stellt eine Reihe von in verschiedenen Programmiersprachen geschriebenen Implementierungen von Benchmark-Algorithmen zur Verfügung (siehe auch Kapitel 3.1 auf Seite 16). Das Hauptziel dabei, die Performanz der einzelnen Programme und damit der Programmiersprachen zu vergleichen.

In Abbildung 3.2 ist die Auswertung der Laufzeitmessungen dieser Testprogramme dargestellt. Zu erkennen ist, dass der (normalisierte) Mittelwert von Rust über alle Benchmarks zwar nahe, aber über dem Vergleichswert von C++ liegt (wobei die Varianz deutlich größer ist). Der Wert von C mit etwa 1,3 noch weiter entfernt und liegt damit außerhalb des oberen Quartils⁴ von Rust. Java ist noch deutlich langsamer: Das untere Quartil entspricht etwa dem Maximum von C und Rust, der Mittelwert ist etwa doppelt so hoch wie bei C und fast dreimal so hoch wie Rust.

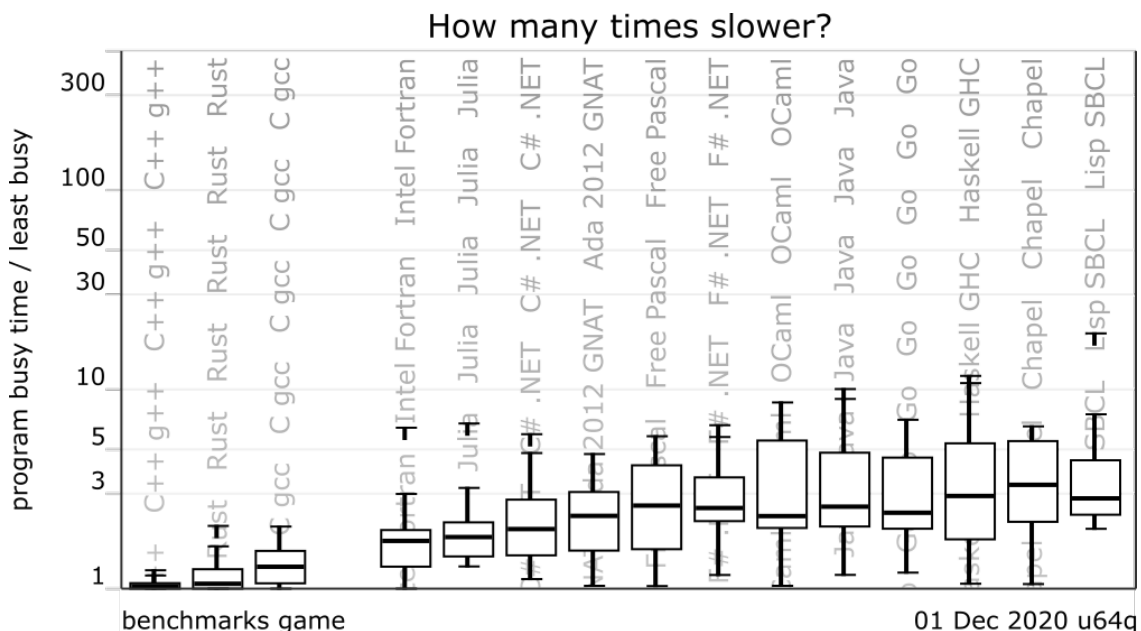


Abbildung 3.2: Vergleich der Berechnungszeiten von Benchmarks für Programmiersprachen (normalisiert), sortiert nach dem geometrischen Mittel
Quelle: [41]

Bei einem Vergleich der Einzelwerte (siehe Abbildung 3.3, S. 21 und Tabellen B.4 bis B.13, S. 78 ff.) ist das Ergebnis ähnlich: Lediglich in einem Benchmark (*pidigits*) ist das schnellste Rust-Programm minimal langsamer als die schnellste C-Variante, in allen anderen Benchmarks schneidet C schlechter ab als Rust – in zwei Fällen allerdings nur knapp (unter 5 % über der Zeit von Rust).

⁴ Das untere und obere Quartil sind die Punkte unter und über dem Median, zwischen denen 50 % der Werte liegen, davon jeweils 25 % zwischen Quartil und Median

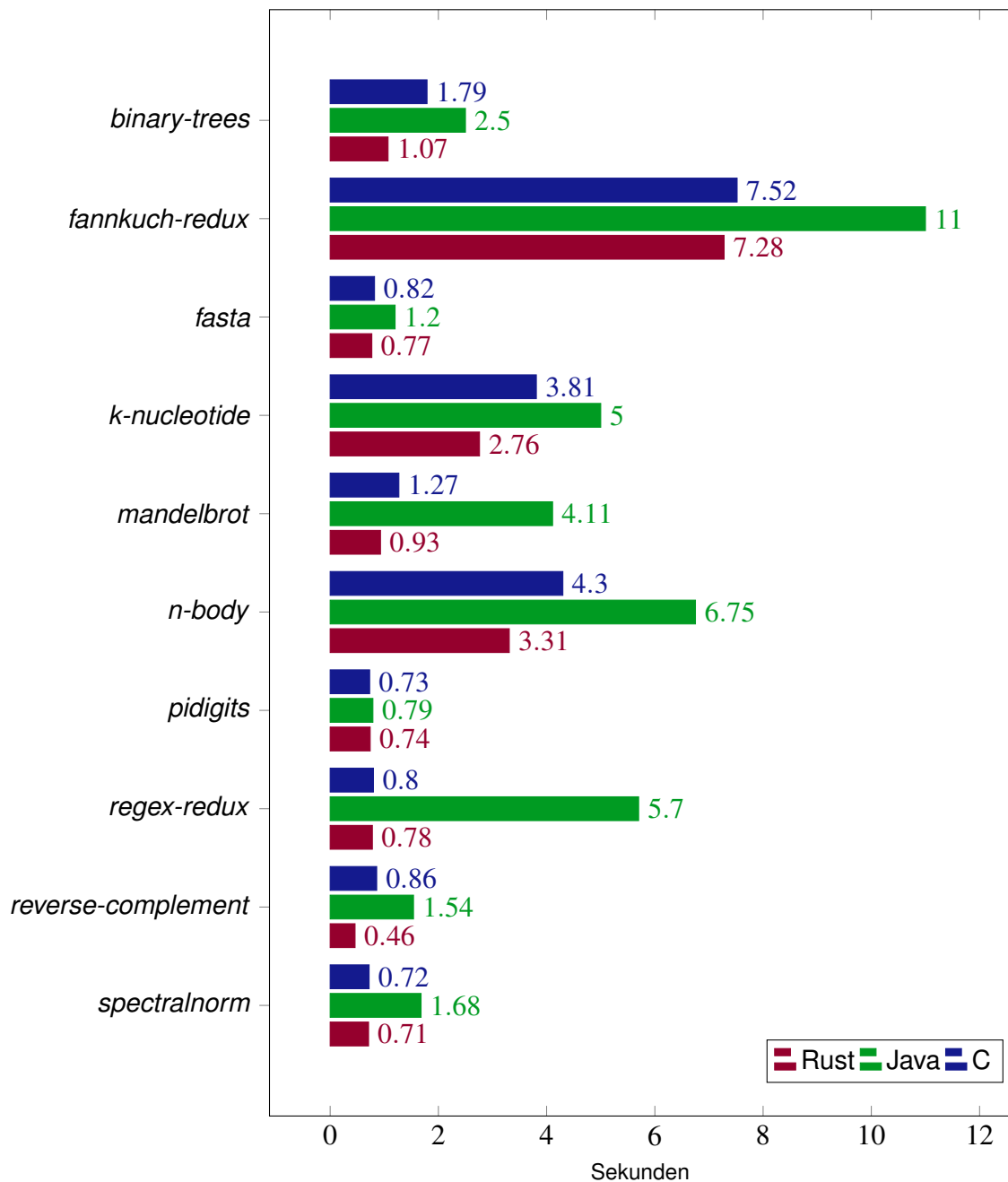


Abbildung 3.3: Berechnungszeiten der Benchmarks pro Algorithmus: schnellste Zeit aus Tabellen B.4 bis B.13 (S. 78 ff.).

Quelle: [37]

3.2.3 Speicherverbrauch

Das *Computer Language Benchmarking Game* (s. Kapitel 3.1) prüft für alle gegebenen Programme nicht nur die Laufzeit, sondern auch die Speichernutzung während der Messung (siehe Abbildung 3.4, S. 22), was ebenfalls ein wichtiger Aspekt bei der Beurteilung eines Programms oder einer Programmiersprache ist.

Dabei schneidet Java erneut in jedem einzelnen Benchmark am schlechtesten ab. Zwischen Rust und C ist das Ergebnis hingegen nicht mehr so klar. Bei sieben von zehn Benchmarks hat C eine bessere Speichernutzung, bei zweien davon nur knapp (Rust unter 5 % über dem Wert von C). Umgekehrt liegt Rust in einem Fall nur knapp vorne, bei *fasta* und *reverse-complement* hingegen deutlich.

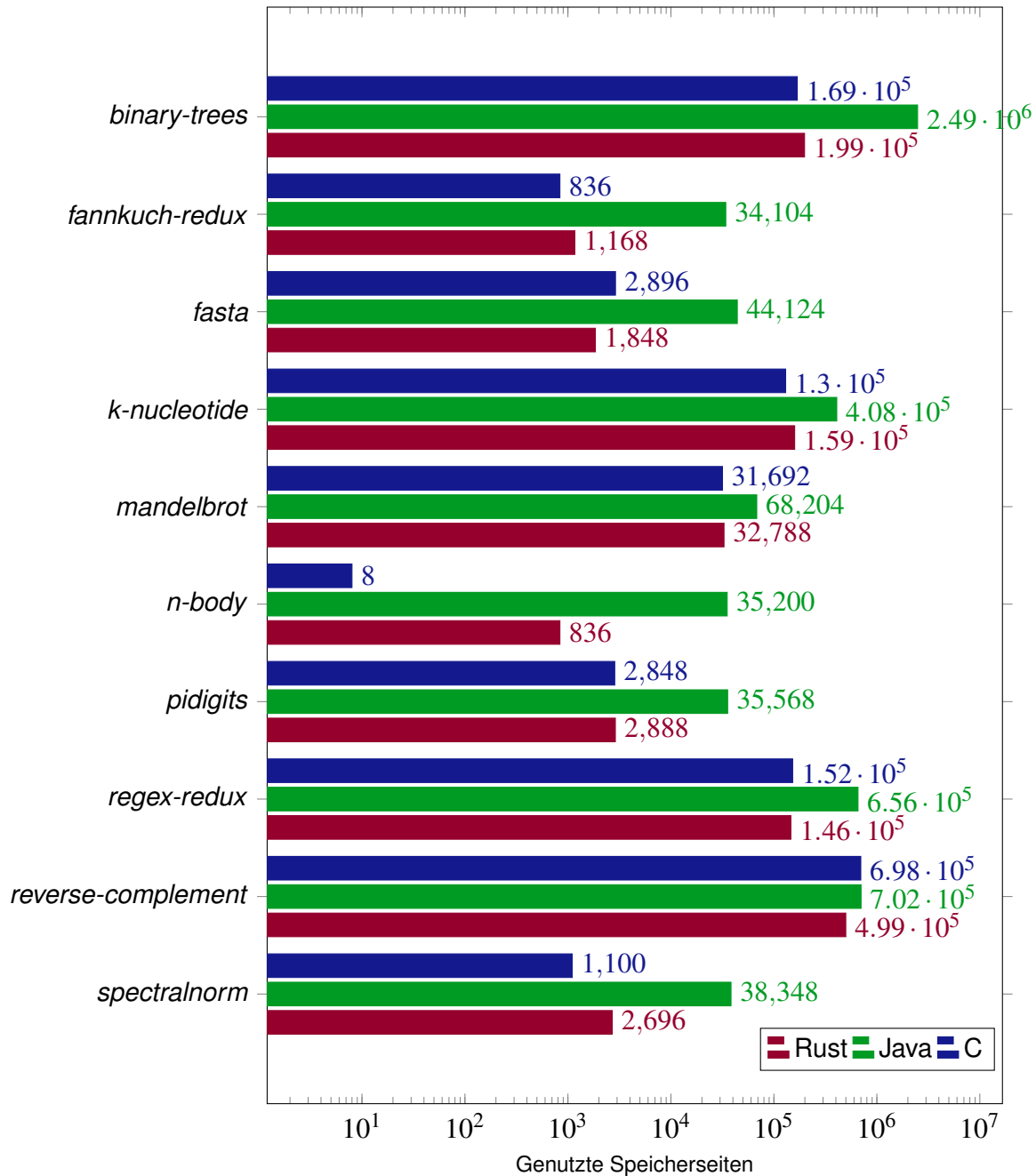


Abbildung 3.4: Speicherverbrauch der Benchmarks pro Algorithmus (in Speicherseiten von je 4 KiB; logarithmische Darstellung): schnellste Zeit aus Tabellen B.4 bis B.13 (S. 78 ff.).

Quelle: [37]

3.3 Sicherheit

Bei der Betrachtung der Sicherheit steht C im Vordergrund: C ist bekannt für viele und schwerwiegende Sicherheitslücken. In diesem Abschnitt werden daher zunächst typische sicherheitsrelevante Fehler in C-Programmen beschrieben und anschließend eine reale Sicherheitslücke analysiert.

3.3.1 Theorie

Die häufigsten Gründe für Sicherheitsprobleme in C-Programmen sind laut [42, S. 846–848] (neben Logikfehlern, welche natürlich immer auftreten können) fehlerhafte Formatierungsfunktionen, Pufferüberläufe, *Race Conditions* und Speicherfehler.

Im Folgenden werden die einzelnen Fehlerarten für C kurz beschrieben und erläutert, ob sie auch in Java oder Rust auftreten können und warum (nicht).

Formatstring-Angriffe

Ein Formatstring-Angriff nutzt einen Programmierfehler aus, bei dem ein von Nutzer*innen kontrollierter Puffer als Formatstring anstatt als Argument an eine Formatierungsfunktion übergeben wird.

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[]) {
4      if (argc < 2) {
5          return -1;
6      }
7
8      // richtig
9      printf("%s\n", argv[1]);
10
11     // falsch
12     printf(argv[1]);
13
14     return 0;
15 }
```

Die Funktionen der *printf*-Familie erwarten (neben anderen Parametern) zuerst einen Formatstring, in dem über sogenannte *Format-Tokens* die Anzahl und Art der nachfolgenden Format-Argumente festgelegt wird. Der Formatstring "%d + %d = %d" erwartet beispielsweise drei Ganzzahl-Argumente. Somit werden die nächsten drei mal vier Bytes⁵ vom Stack gelesen und ausgegeben werden. Im Normalfall befinden sich dort die entsprechenden Argumente.

Falls Angreifer*innen es allerdings schaffen, den Formatstring zu kontrollieren, können darüber beliebige Werte ausgegeben werden. Wenn dem Programm im Beispiel oben %d als erstes Argument übergeben wird, gibt es die nächsten vier Bytes auf dem Stack als Dezimalzahl aus, %x als Hexadezimalzahl. Mit mehreren Parametern lassen

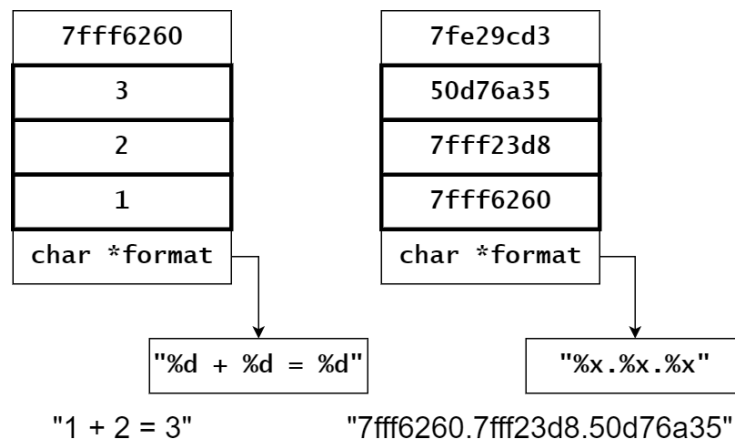


Abbildung 3.5: Aufruf *printf*: Korrekt mit Argumenten und falsch mit nutzerkontrolliertem Formatstring

sich große Teile des Stacks auslesen (vgl. Abbildung 3.5).

Interessant ist auch der Format-Token %n: Für diesen wird als Argument eine Adresse erwartet, in die beim Aufruf der Funktion die Anzahl der bisher ausgegebenen Bytes geschrieben wird.

```

1  #include <stdio.h>
2
3  int main() {
4      int bytes;
5      printf("hello, world%n\n", &bytes);
6      printf("%d bytes written\n", bytes); // 12
7      return 0;
8  }
```

Angreifer*innen können in vielen dieser Fälle sowohl diese Anzahl als auch die Adresse kontrollieren. Damit ist es mit etwas Aufwand möglich, beliebige Werte von und in beliebige/n Speicheradressen zu lesen bzw. schreiben. [43, S. 183–212], [42, S. 847]

⁵ je nach Plattform kann der *int*-Datentyp auch größer oder kleiner als vier Bytes sein

Fehler dieser Art lassen sich mit einer statischen Analyse relativ einfach entdecken – `gcc` gibt eine Warnung aus, wenn mit dem Flag `-Wformat-security` kompiliert wird – und kommen daher inzwischen nicht mehr so oft vor. Falls mit einem entsprechenden Tool kein solcher Fehler gefunden wird, ist es sogar sicher, dass keiner vorhanden ist. Das gilt für die meisten anderen Sicherheitslücken nicht. [44, S. 9]

Der Grund für dieses Problem sind sogenannte *variadische Funktionen*, die eine unbestimmte Zahl von Parametern haben. In C wird dabei als erstes Argument oft die Anzahl der folgenden Argumente erwartet. Allerdings werden weder deren Anzahl noch die Typen überprüft; in der Funktion muss sich darauf verlassen werden, dass die Aufruf-Funktion die richtige Anzahl von Argumenten des richtigen Typs übergibt.

Innerhalb der Funktion werden dann entsprechend viele Werte vom Stack abgerufen, unabhängig davon, ob dort tatsächlich Argumente oder andere Daten stehen.

In Java existieren ebenfalls Funktionen mit variabler Argumentzahl, kurz *Varargs*. Diese Argumente werden allerdings vom Compiler in ein Array umgewandelt und so als einzelnes Argument übergeben.

Da die Länge eines Arrays in Java bekannt ist, entfällt die Notwendigkeit eines zusätzlichen Parameters. Falls es diesen wie bei der `printf`-Funktion indirekt trotzdem gibt, kann die Funktion relativ einfach prüfen, ob genügend Elemente vorhanden sind. Im schlimmsten Fall wird auf ein nichtexistierendes Array-Element zugegriffen, was zu einer Exception führt. [21, S. 282–285]

```
Varargs-Funktion in Java
1 public class FormatExploit {
2     public static void main(String[] args) {
3         if (args.length < 1) {
4             return;
5         }
6         // richtig
7         System.out.printf("%s%n", args[0]);
8         // falsch
9         System.out.printf(args[0]);
10    }
11 }
```

Wenn dieses Programm mit dem Argument `%d` aufgerufen wird, stürzt es mit folgender Exception ab:

```
Exception in thread "main" java.util.MissingFormatArgumentException:
Format specifier '%d'
```

In Rust gibt es solche Funktionen nicht. Formatierer werden als Makro umgesetzt und beim Kompilieren geprüft. Das `format_args`-Makro, welches von allen anderen Makros für die Formatierung genutzt wird, definiert als ersten Parameter ein *string literal*, also einen im Code festgelegten und zur Kompilierzeit feststehenden String. Diese Voraussetzung trifft auf sämtliche nutzerkontrollierte Strings nicht zu. Der Versuch, den obigen Code in Rust zu schreiben, führt zu einer Fehlermeldung und dem Abbruch des Kompilervorgangs:

```

1  use std::env;
2
3  fn main() {
4      let mut iter = env::args();
5      let _name = iter.next();
6      let first_arg = iter.next();
7      if first_arg.is_none() {
8          return;
9      }
10
11     let first_arg = first_arg.unwrap();
12     // richtig
13     println!("{}", first_arg);
14     // Compiler error
15     println!(first_arg);
16 }

```

```
error: format argument must be a string literal
```

```
--> src\main.rs:15:14
```

```

|
15 |     println!(first);
|               ^^^^^
|

```

```
help: you might be missing a string literal to format with
```

```

|
15 |     println!("{}", first);
|               ^^^^^

```

```
error: aborting due to previous error
```

Pufferüberlauf

Bei einem Pufferüberlauf werden in einen Puffer der Größe n mehr als n Bytes geschrieben. Dies kann manuell passieren oder durch Aufruf einer Funktion wie *gets* oder *strcpy*, die eine Nutzereingabe bzw. einen Speicherbereich in einen Puffer kopieren, ohne allerdings die Größe der beiden zu prüfen.

Ein einfaches Beispiel zeigt dieses Verhalten in C:

```
Beispiel C-Programm mit Pufferüberlauf
1  #include <stdio.h>
2  #include <string.h>
3
4  int check_passwd(char password[]) {
5      char passwd_buffer[100];
6      int password_correct = 0;
7
8      strcpy(passwd_buffer, password);
9
10     if (strcmp(passwd_buffer, "secret") == 0) {
11         password_correct = 1;
12     } else if (strcmp(passwd_buffer, "password") == 0) {
13         password_correct = 1;
14     }
15
16     return password_correct;
17 }
18
19 int main(int argc, char *argv[]) {
20     if (argc < 2) {
21         printf("Usage: %s <password>\n", argv[0]);
22         return -1;
23     }
24
25     if (check_passwd(argv[1])) {
26         printf("Password accepted\n");
27     } else {
28         printf("Wrong password\n");
29     }
30     return 0;
31 }
```

Wie erwartet werden *secret* und *password* akzeptiert, *1234* hingegen nicht. Wenn das Programm allerdings beispielsweise mit 110 'A's aufgerufen wird, gibt es ebenfalls die Erfolgsmeldung aus.

Dabei handelt es sich um einen Pufferüberlauf: Die *strcpy*-Funktion in Zeile 8 kopiert den gesamten übergebenen String in den Ziel-Puffer *passwd_buffer*. Da dieser nur 100 Byte groß ist, wird dabei auch die im Speicher dahinter liegende Variable *password_correct* überschrieben.

Und weil C in *if*-Abfragen nur zwischen „0“ und „nicht 0“ unterscheidet, wird der neue Wert als *wahr* gewertet.

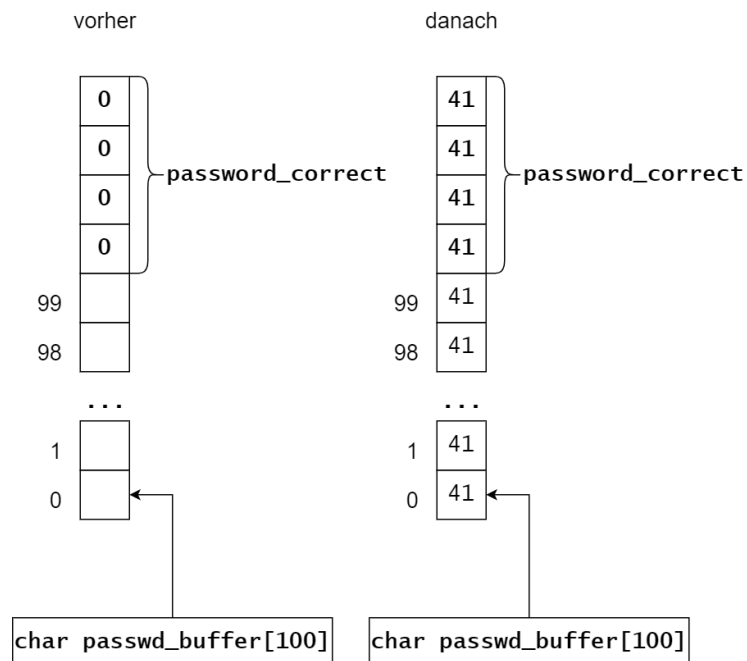


Abbildung 3.6: Pufferüberlauf auf dem Stack (0x41 = 'A')

In diesem Fall würde es ausreichen, die *strcpy*-Funktion in Zeile 8 durch *strncpy* zu ersetzen und die Größe des Puffers als zusätzliches Argument zu übergeben.

Weder in Rust noch in Java ist der direkte Zugriff auf Speicheradressen möglich. Zugriffe auf Array-Elemente werden in beiden Sprachen geprüft; ein Zugriff außerhalb der Array-Grenzen führt in Java zu einer *ArrayIndexOutOfBoundsException*, in Rust zu einer *panic*.

In Java ist diese Prüfung auch nicht abschaltbar, allerdings wird sie zur Laufzeit automatisch entfernt, wenn ein Zugriff auf ein Element außerhalb der Array-Grenzen nicht möglich ist [21, S. 58].

In Rust kann ein *unsafe*-Block genutzt werden, um die Prüfung zu umgehen. Allerdings ist das eine bewusste Entscheidung, die nur getroffen werden sollte, wenn auf eine andere Weise geprüft wird, dass der Zugriff innerhalb der Grenzen erfolgt.

Race Conditions

Der Begriff *race condition* wird oft im Zusammenhang mit *data races* genutzt, teilweise werden beide Bezeichnungen synonym verwendet.

Bei *data races* handelt es sich um den gleichzeitigen Zugriff von mehreren Punkten auf eine Variable bzw. eine Speicheradresse. Es könnte beispielsweise ein Thread eine Zahl in eine Speicherzelle schreiben, während ein anderer Thread die Zahl aus dieser Speicherzelle liest. Das Ergebnis ist kaum vorhersehbar. Der zweite Thread könnte die vollständige Zahl vor oder nach dem Schreibvorgang des ersten Threads lesen, es ist aber auch möglich, dass der gelesene Wert aus Bits des vorherigen und des neuen Werts besteht. *Data races* können in C oder Java verhindert werden, indem der gleichzeitige Zugriff auf eine Ressource durch ein Synchronisationswerkzeug wie ein Mutex⁶ blockiert wird oder indem atomare Operationen⁷ verwendet werden.

Durch das *Ownership*-Prinzip sind *data races* in Rust fast ausgeschlossen [45, Kap. 8.1].

Race conditions sind hingegen Situationen, deren Ausgang sich durch verschiedene Ausführungsreihenfolgen ändert: Das Ergebnis, wenn Thread 1 vor Thread 2 ausgeführt wird, ist anders, als wenn mit Thread 2 begonnen wird [42, S. 846].

Ein Beispiel für so eine Situation: Zwei Threads versuchen gleichzeitig, eine Variable mit Startwert 1 um 1 zu erhöhen (vgl. Abbildung 3.7; *data races* werden hierbei nicht beachtet). Je nach Ausführungsreihenfolge ist der gespeicherte Wert danach 2 oder 3.

Race conditions können in allen drei Sprachen auftreten; eine allgemeingültige Empfehlung zur deren Vermeidung existiert nicht. Gängige Maßnahmen sind ebenfalls Synchronisationsverfahren wie Semaphore⁸ oder Mutexe.

Im Beispiel müssten beide Threads vor dem ersten Auslesen eine Sperre einrichten und den Zugriff erst nach dem Zurückschreiben wieder freigeben. Damit müsste der zweite Thread warten, bis der erste fertig ist, und könnte erst dann den bereits inkrementierten Wert auslesen. Auf diese Weise wäre sichergestellt, dass am Ende stets das richtige Ergebnis gespeichert würde.

⁶ Ein *Mutex* (von *Mutual exclusion*, gegenseitiger Ausschluss) erlaubt einen einzelnen Zugriff auf ein Element und blockiert weitere Zugriffe, solange der erste nicht abgeschlossen ist

⁷ *atomare* (unteilbare) Operationen bestehen aus einem einzelnen Schritt und können so nicht teilweise ausgeführt sein

⁸ Semaphore bestehend grundlegend aus einem Zähler und zwei Funktionen, die diesen Zähler erhöhen oder reduzieren. Falls der Zähler 0 erreicht, werden weitere Aufrufe der zweiten Funktion blockiert, bis er wieder höher ist.

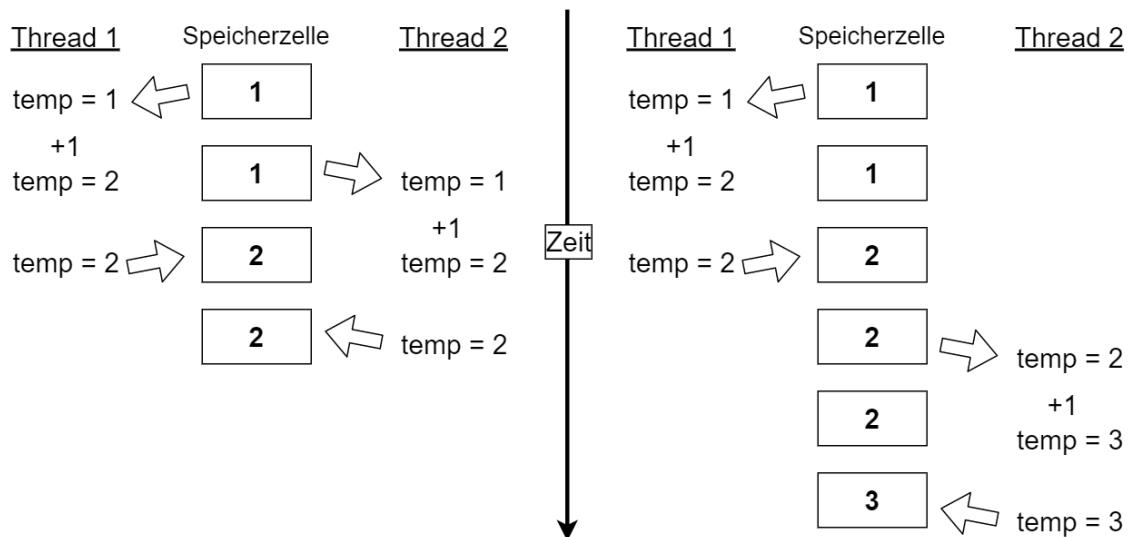


Abbildung 3.7: Beispiel einer *Race condition*: mögliche Ausführungsreihenfolgen

Speicherfehler

Es gibt verschiedene Arten von Fehlern, die alle die gleiche Ursache haben: Fehler in der Speicherverwaltung. Insbesondere die Freigabe von Speicher, beispielsweise per Aufruf von *free*, ist problembehaftet. Häufige Quellen von Sicherheitslücken sind die Nutzung eines Speicherbereichs nach dessen Freigabe (*use after free*), die doppelte Freigabe eines Bereichs (*double free*), oder das Fehlen der Freigabe für einen Speicherbereich (Speicherleck, *memory leak*).

Speicherleck Bei einem Speicherleck wird Speicher angefordert und nicht wieder freigegeben. Ein möglicher Grund sind Fehlerbehandlungen, die zu einer vorzeitigen Beendigung einer Funktion führen, oder Unklarheit darüber, welche Funktion zuständig ist für die Freigabe. Über einen langen Zeitraum hinweg kann sich so der Hauptspeicher des ausführenden Computers füllen. Die mögliche Folge eines Angriffes, der diesen Mechanismus ausnutzt, ist der Ausfall von Programmen oder Systemen (*Denial of Service*), was beispielsweise im Fall einer Firewall weitere Angriffe ermöglichen kann. Abgesehen davon gibt es für Angreifer kaum Möglichkeiten, dieses Problem auszunutzen. [46], [47]

Während es keine realistische Möglichkeit gibt, Speicherlecks generell zu verhindern, versuchen sowohl Java als auch Rust, deren Auftreten zu reduzieren. In Java sorgt der *Garbage Collector* dafür, dass nicht mehr genutzter Speicher freigegeben wird. Aktiv wird er für Speicherbereiche, auf die es keine Referenz mehr gibt. Klassenvariablen können die Bereinigung aber verhindern, da sie während der Programmlaufzeit nicht gelöscht werden und so immer einen Verweis auf den zugehörigen Speicher existiert. Effektiv ist das in Rust ähnlich, nur dass der Speicher direkt beim Löschen (*drop*) der

Variablen (beispielsweise beim Verlassen des Gültigkeitsbereichs) freigegeben wird und nicht erst zu einem unbestimmten späteren Zeitpunkt wie mit *Garbage Collector*.

Double free *Double free* bezeichnet die mehrfache Freigabe eines Speicherbereichs. Dies kann beispielsweise passieren, wenn nicht klar ist, wo bzw. in welcher Funktion dieser Vorgang passieren soll. Eine wahrscheinliche Folge ist die Korruption der Datenstrukturen, die für die Speicherverwaltung genutzt werden. Dadurch kann das Programm abstürzen, oder es kann zu einem späteren Zeitpunkt für zwei Aufrufe von *malloc* der gleiche Speicherbereich zurückgegeben werden, was im Fall eines Angriffes vielfältige Folgen bis zur Ausführung von beliebigem Code haben kann. [48], [49]

In Java gibt es keine mit *free* vergleichbare Funktion, da sich der *Garbage Collector* um die Freigabe von Speicher kümmert.

Bei Rust sorgt das *Ownership*-Prinzip dafür, dass der Destruktor und damit auch die Speicherfreigabe immer nur einmal für jedes Element läuft. Neben der automatischen Freigabe nach Ende des Gültigkeitsbereichs gibt es noch die Funktion *drop*, über die ein Element zu einem beliebigen Zeitpunkt vernichtet werden kann. Das Element muss dafür in die Funktion verschoben werden und ist danach effektiv verschwunden. Es kann also nicht mehr genutzt werden, ohne einen Compilerfehler zu generieren. Das schließt sowohl das manuelle Freigeben per *drop* als auch das automatische ein.

Use after free Mit *Use after free* oder auch *Dangling Pointer* (etwa: baumelnder Zeiger) wird die Nutzung eines Speicherbereichs nach dessen Freigabe beschrieben.

Normalerweise wird der Speicher, nachdem er durch das Programm freigegeben wurde, nicht sofort wieder genutzt, sondern er bleibt noch einige Zeit im gleichen Zustand wie zuvor. Dann kann auch der Inhalt des Speichers noch identisch sein wie zum Zeitpunkt vor der Freigabe. Daher führt die weitere Nutzung nicht unbedingt zu einem Absturz des Programmes, wie es bei einem Zugriff auf einen nicht zugewiesenen Speicherbereich passieren würde. Teilweise gibt es keinen Hinweis, dass sich irgendetwas geändert hat, was das Identifizieren eines solchen Fehlers kompliziert macht.

Allerdings ist es auch möglich, dass zum Zeitpunkt der erneuten Nutzung schon andere Daten in dem entsprechenden Bereich gespeichert sind. Das können beispielsweise Variablen sein oder Zeiger auf Funktionen. Je nach Art des Zugriffes kann das Programm abstürzen, es können falsche Daten gelesen, fremde Daten überschrieben oder (hauptsächlich bei C++) sogar ungewollte Funktionen ausgeführt werden. Angreifer*innen können darüber unter Umständen eigene Funktionen ausführen und so die Kontrolle über das Programm und ggf. das System übernehmen. Damit handelt es sich um eine schwerwiegende Sicherheitslücke. [50], [51]

Java und Rust sind aus den gleichen Gründen wie bei *double free* nicht betroffen.

3.3.2 Beispiel einer Sicherheitslücke: *openssl-too-open*

Die praktischen Auswirkungen der fehlenden Sicherheitsfunktionen in C lassen sich am besten an in den letzten Jahren entdeckten Sicherheitslücken erkennen. In diesem Abschnitt wird eine von ihnen vorgestellt.

Bei *openssl-too-open* (CVE-2002-0656⁹; CVSS-Base-Score¹⁰: 7,5 [52]) handelt es sich um einen Pufferüberlauf in der *OpenSSL*-Bibliothek, der zu *Remote Code Execution* (RCE) auf dem Server führen konnte.

Während der Aushandlung der SSL-Parameter kann der Client ein zusätzliches Argument (*KEY_ARG*) mitsenden, das je nach Verschlüsselungsalgorithmus unterschiedlich interpretiert werden kann. Der Client sendet dabei zuerst die Länge des Arguments und anschließend das Argument selbst. Auf dem Server werden nun so viele Bytes wie angegeben von dem Argument in einen acht Byte großen Puffer kopiert. Falls das Argument länger ist, kommt es zu einem Überlauf und es werden Parameter in der Struktur *ssl_session_st*, in der die Einstellungen verwaltet werden, überschrieben (siehe Abbildung 3.8: Der gesamte rot umrandete Bereich kann mit neuen Daten gefüllt werden).

Beim Aufbau einer SSL-Verbindung können so mehrere Werte und Zeiger verändert werden. Damit ist es möglich, zuerst die Adresse der Datenstruktur im Speicher herauszufinden, indem eine weitere Variable, die die Länge eines Feldes (Session-ID) speichert, mit einem höheren Wert überschrieben wird (in der Abbildung blau dargestellt). Der Server überträgt so in seiner Antwort nicht nur dieses Feld, sondern auch alle folgenden Daten (blau umrandet) und insbesondere einen Zeiger, mit dem sich die Adresse der Struktur berechnen lässt: Er zeigt genau 368 Bytes hinter den Beginn. Im nächsten Schritt kann hier *Shellcode*¹¹ platziert werden. Anschließend kann durch Ausnutzen der *free*-Funktion dafür gesorgt werden, dass die Adresse der *free*-Funktion in der *Global Offset Table* (GOT)¹² mit der Adresse des Shellcodes überschrieben wird. Hierfür ist die Strukturierung der Daten als verkettete Liste hilfreich: *free* greift auf die Adresse des vorherigen Elements (-1) zu und setzt dort den Nachfolger (+1) des aktuellen Elements (0) als nächstes Element (vorher: -1 -> 0 -> 1; nachher: -1 -> 1). Da bei dem Pufferüberlauf aber die beiden Zeiger auf das vorherige und nächste Element über-

⁹ Sicherheitslücken wird normalerweise durch die MITRE-Organisation eine CVE-Nummer (Common Vulnerabilities and Exposures) zugewiesen. MITRE wird von der US-Regierung finanziert und betreibt Forschungsinstitute in den USA.

¹⁰ Das *Common Vulnerability Scoring System* dient der Bewertung und Gewichtung von Sicherheitslücken. Eine Lücke wird als gefährlicher gewertet, wenn sie leichter auszunutzen ist oder der potenzielle Schaden größer ist. Die Werte reichen von 0 bis 10.

¹¹ *Shellcode* bezeichnet eine Menge von Maschinenbefehlen, die einen Kommandozeileninterpreter (*Shell*) starten oder es auf andere Weise ermöglichen, Befehle auf dem fremden System auszuführen

¹² Die *Global Offset Table* enthält die Adressen der Funktionen im Programm. Bei Aufruf einer Funktion wird die Adresse dieser Funktion aus der GOT abgerufen und die Programmausführung nach einem Sprung dort fortgesetzt.

geschrieben werden können, kann darüber ein beliebiger Wert in eine beliebige Adresse geschrieben werden – beispielsweise die Adresse des Shellcodes in die GOT. Bei dem nächsten Aufruf der Funktion wird der Shellcode aufgerufen und ausgeführt. [53]

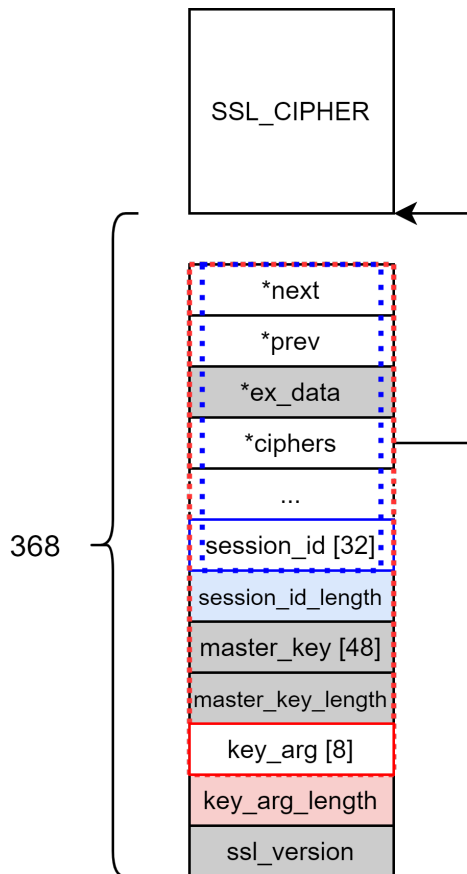


Abbildung 3.8: Speicherlayout bei *openssl-too-open*: Durch eine falsche *key_arg_length* kann mit einem Pufferüberlauf von *key_arg* der rote Bereich überschrieben werden; mit veränderter *session_id_length* kann der blaue Bereich ausgelesen werden

Bei diesem Angriff handelt es sich um eine geschickte Kombination verschiedener Sicherheitslücken, die bei erfolgreicher Ausnutzung die Ausführung von beliebigem Code auf dem angegriffenen System ermöglichen – eine der schlimmsten anzunehmenden Auswirkungen eines Angriffs.

Die Ursache für den Fehler war die fehlende Prüfung, ob die Länge des Arguments kleiner oder gleich der Größe des Puffers ist. Ein solcher Fehler kann selbst erfahrenen Programmierern unterlaufen und lässt sich auch in anderen Sprachen nicht wirksam verhindern.

Für die Folgen gilt das allerdings nicht: Abgesehen davon, dass es in Java und Rust die benötigten Strukturen wie Zeiger nicht oder nicht in dieser Weise gibt und dieser spezielle Angriff so nicht möglich wäre, sind auch die grundsätzlichen Probleme nicht vorhanden. Sowohl das Schreiben über die Grenzen des *key_arg*-Feldes als auch das Lesen über *session_id* würden zu einer *Exception* (Java) bzw. *panic* (Rust) führen. Es wäre für Angreifer*innen weder möglich, andere Daten als gewollt zu ändern, noch wichtige Daten wie Speicheradressen auszulesen.

4 Prototypischer Webserver als Vergleichsobjekt

Die theoretischen Gemeinsamkeiten und Unterschiede der drei Sprachen sollen auch praktisch – anhand eines Programmes – analysiert werden. Ziel ist es hierbei, dasselbe Programm in jeder der drei Sprachen so einfach wie möglich unter Einhaltung der jeweiligen Sprachstandards zu implementieren.

Die Wahl fällt dabei auf einen einfachen Webserver. Zunächst wird beschrieben, was einen solchen ausmacht, um anschließend Anforderungen an die Programme stellen zu können. Die fertiggestellten Programme werden schließlich kurz vorgestellt.

4.1 Webserver

Ein Webserver ist ein Server, der Daten über das *Hyper Text Transfer Protocol* (HTTP) an Clients wie Webbrowser überträgt. Ganz einfach zusammengefasst, liefert er eine Webseite an anfragende Programme aus.

In diesem Abschnitt wird beschrieben, wie genau ein Webserver diese Aufgabe erfüllt und warum ein solcher für den Vergleich genutzt wird.

4.1.1 Historie

Der erste Webserver wurde 1990 zeitgleich mit dem ersten Webbrowser von Tim Berners-Lee geschaffen [54, S. 192], [55] und war unter dem Namen *CERN httpd* bekannt [56].

Der meistbenutzte Webserver war über lange Zeit (und ist es bestimmten Zählungen nach auch heute noch [57]) der *Apache HTTP Server*¹³ bzw. dessen Vorgänger *NCSA HTTPd*. Inzwischen wurde er von *nginx*¹⁴ eingeholt, danach folgen *Microsoft IIS* und *Google Web Server*. [58]

¹³ <https://httpd.apache.org/>

¹⁴ <https://nginx.org/>

4.1.2 Technik

Die Aufgabe eines Webserver besteht darin, eine URL in einen Dateinamen zu übersetzen und diese Datei dann über das Internet zurückzuschicken oder sie in einen Programmnamen zu übersetzen, das Programm auszuführen und dessen Ausgabe zurückzugeben. Das ist der Kern dessen, was ein Webserver macht: der Rest ist Zierrat. ([59, S. 1])

Der Client – in der Regel ein Webbrowser – sendet eine Nachricht an einen Computer, auf dem der Webserver läuft (siehe Abbildung 4.1). Als Protokoll wird dabei HTTP genutzt, in der Regel in der per SSL/TLS verschlüsselten Variante (HTTPS). Diese Nachricht besteht mindestens aus einer HTTP-Methode, einem Pfad und der benutzten HTTP-Version. Zusätzlich können noch HTTP-Header und ggf. Daten (der *body*) folgen. Eine Anfrage könnte beispielsweise so aussehen:

```
GET /index.html HTTP/1.1
Host: example.com
```

Dabei ist *GET* die HTTP-Methode und */index.html* der Pfad. *HTTP/1.1* steht für die benutzte Version. Der nachfolgende *Host*-Header ist bei HTTP 1.1 vorgeschrieben.

Die zugehörige, vom Client aufgerufene URL lautet *http://example.com/index.html*: Aus ihr erzeugt der Browser die obige Anfrage an den Webserver. Die Bezeichnung der Komponenten in einer URL ähneln denen der Nachricht: *http* wird das *Schema* genannt, *example.com* ist der *Host* und */index.html* der *Pfad*. [59, S. 1–2]

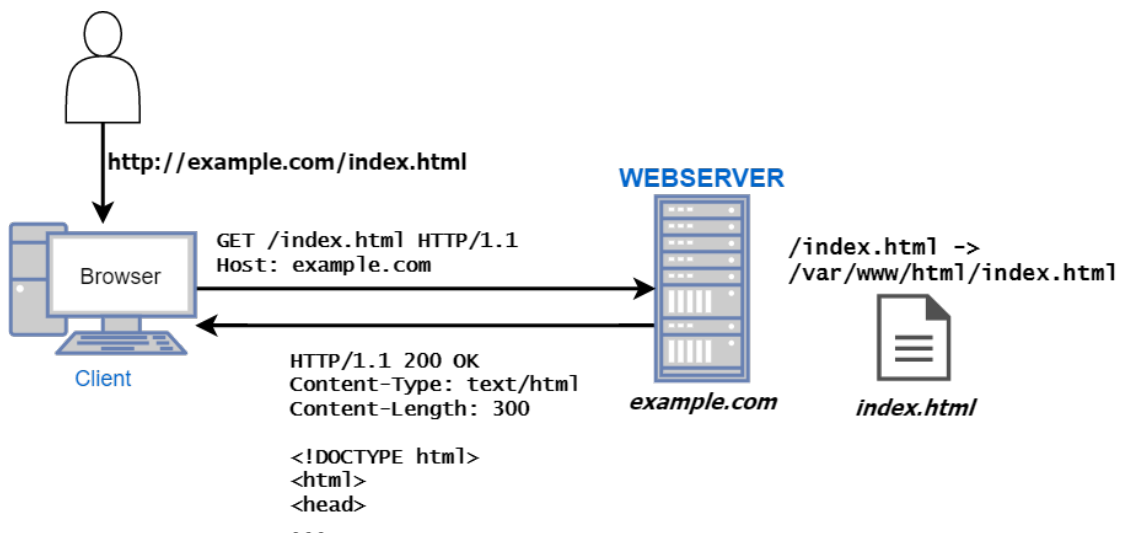


Abbildung 4.1: Abruf einer Datei von einem Webserver

Die Aufgabe des Webserver ist es nun, aus dieser Anfrage eine Antwort für den Client zu generieren.

Es gibt verschiedene Anfragetypen bzw. HTTP-Methoden. Die häufigsten orientieren sich dabei am CRUD-Modell: Daten können erstellt (**Create**; *PUT* oder *POST*), gelesen (**Read**; *GET*), aktualisiert/verändert (**Update**; *PATCH* oder *PUT*) oder gelöscht (**Delete**; *DELETE*) werden [60].

Dateiübertragung

Das Versenden von Dateien war der ursprüngliche Zweck eines Webserver.

Der Prozess kann in drei Schritte aufgeteilt werden: Zunächst wird der relative Anfragepfad in einen absoluten Pfad auf dem Servercomputer aufgelöst. Dabei bezieht sich *relativ* auf das Serververzeichnis, was im Webkontext allerdings oft als absoluter Pfad bezeichnet wird, um eine Unterscheidung zu Pfaden relativ zum aktuellen Verzeichnis wie *../index.html* zu bieten. Für das Beispiel oben könnte aus */index.html* */var/www/html/index.html* werden.

Dieser Pfad wird im zweiten Schritt verschiedenen Prüfungen unterzogen: Existiert eine solche Datei¹⁵ überhaupt? Befindet sie sich im Serververzeichnis? Ist sie aus irgendeinem Grund nicht lesbar für den Serverprozess? Hat der Client die Berechtigung, darauf zuzugreifen?

Falls es sich bei der Datei um einen Ordner handelt: Was soll passieren? Soll eine bestimmte Datei (wie *index.html*) aus diesem Verzeichnis benutzt werden? Oder soll eine Auflistung der Dateien in dem Verzeichnis zurückgegeben werden? Oder soll der Client mit einer Fehlermeldung auf diesen Umstand aufmerksam gemacht werden?

Im dritten Schritt werden aus der Datei die notwendigen Metadaten (beispielsweise die Größe der Antwort und der Dateityp) erzeugt und zusammen mit dem Dateiinhalt an den Client gesendet, sofern alle Prüfungen erfolgreich waren. Ansonsten kann eine Fehlermeldung (falls möglich mit entsprechender HTML-Seite) zurückgegeben werden. [61, S. 35–36]

Ausführung von Programmen

Relativ schnell wurde klar, dass auch dynamische Webseiten benötigt werden. Für eine solche wird durch die Anfrage serverseitig ein Programm gestartet, welches – eventuell durch Auswertung der Anfrage – eine Seite generiert, die dann an den Client zurückgegeben wird.

¹⁵ *Datei* bezeichnet im Folgenden normalerweise die Linux-Variante, die Ordner einschließt

Der Webserver stellt dem aufzurufenden Programm hierbei eine Reihe von Variablen, die Informationen über den Host enthalten, und vom Client über URI oder Anfragekörper übergebene Parameter bereit.

Aktuell werden hierfür hauptsächlich PHP und ASP.NET eingesetzt [62].

4.1.3 Eignung als Vergleichsobjekt

In diesem Abschnitt sollen kurz die Gründe dargelegt werden, die zur Wahl eines Webserver als Untersuchungsgegenstand geführt haben.

Netzwerk

Kaum ein Programm kommt heute noch ohne Netzwerkzugriff aus. Aktualisierungen, Zusatzprogramme (Plugins), das Senden von Fehlermeldungen an Entwickler – all das benötigt selbst bei eigentlich offline funktionierenden Programmen eine Internetverbindung. Daher ist es bei der Wahl der Programmiersprache eine relevante Frage, wie und wie gut Netzwerkfunktionen unterstützt werden.

Ein Webserver als Programm, dessen Hauptaufgabe es ist, Verbindungen über das Internet herzustellen, ist daher gut geeignet als Demonstration der Netzwerkfunktionen der Programmiersprachen.

Parallelisierung

Aktuelle Computer haben mehrere bis viele CPU-Kerne. Selbst die billigsten Prozessoren haben in der Regel vier logische Kerne, und 64 logische Kerne in einem PC sind möglich, wenn auch vermutlich eher selten. Programme aller Art sollten diese Möglichkeit ausnutzen und Berechnungen auf mehrere Prozessorkerne und Threads/Prozesse aufteilen. Allerdings ist das je nach Programm nicht unbedingt möglich.

Ein Webserver hingegen kann sehr gut geteilt werden: Für jede Verbindung kann ein neuer Thread genutzt werden. Im Gegensatz zu manch anderen Berechnungen existiert auch keine Grenze nach oben, ab der sich Parallelisierung nicht mehr lohnt: Jeder zusätzliche Kern bedeutet eine weitere gleichzeitige Verbindung.

Damit kann gut nachvollzogen werden, welche Funktionalität eine Sprache in Bezug auf Multithreading bietet und wie performant sie dabei ist.

4.2 Anforderungen

Ein aktueller Webserver wie *Apache* oder *nginx* ist das Ergebnis jahrelanger Arbeit vieler Entwickler*innen; ein vergleichbares Programm ist natürlich in kurzer Zeit nicht erreichbar. Daher wird sich hier auf einen kleinen Teil der Funktionen beschränkt, so dass die Grundfunktionalität eines Webserver hergestellt wird, ohne dass der Aufwand zu groß ist. Die Mindestanforderungen sind im Folgenden genauer beschrieben.

4.2.1 Generelle Anforderungen

Zunächst gibt es eine Reihe von grundlegenden Anforderungen an die entstehenden Programme.

Vermeidung von Fremdbibliotheken

Um eine Vergleichbarkeit der Programmiersprachen zu ermöglichen, sollen Bibliotheken, die nicht zur Sprache selbst gehören, nicht verwendet werden, falls es eine Alternative gibt. Diese tragen aus Sicht des Autors nicht dazu bei, Unterschiede zwischen den Sprachen selbst aufzuzeigen, sondern eher die des Ökosystems um die Sprachen. Dies ist zwar durchaus auch ein wichtiger Aspekt bei der Wahl der Sprache, soll aber hier nicht betrachtet werden. Aus Zeit- und Aufwandsgründen kann dies nicht an jeder Stelle verwirklicht werden, aber es wird stets versucht, die Sprache so direkt wie möglich zu nutzen.

4.2.2 Grundlegende Funktionalität

Konfigurationsdatei

Eine grundlegende Anforderung ist die Konfigurierbarkeit des kompilierten Programmes mittels einer Konfigurationsdatei. Um den zu programmierenden Parser zu vereinfachen, und weil die Notwendigkeit fehlt, wird dabei nicht auf ein übliches Format wie XML oder JSON gesetzt, sondern auf einfache Key-Value-Speicherung. In jeder Zeile steht dabei der Datentyp (entweder *int* für Zahlen oder *string* für Zeichenketten), gefolgt von einem Namen (es gibt eine festgelegte Menge von validen Namen) und einem zugewiesenen Wert. Zeilen, die mit *#* beginnen, sollen als Kommentar ignoriert werden. Eine beispielhafte Datei ist in Listing 4.1 zu sehen.

```
1  # port for http server
2  int PORT_HTTP = 3080
3  # website root
4  string WEBSITE_ROOT = "/var/www/html"
```

Listing 4.1: Beispiel-Konfiguration

Verzeichnisansicht

Beim Aufruf eines Pfades ohne Dateinamen (Beispiel: `http://example.com/` anstatt `http://example.com/index.html`) soll der Client eine Verzeichnisübersicht angezeigt bekommen. Diese Übersicht muss zur Laufzeit aus den aktuellen Verzeichnisinhalten generiert und als HTML-Datei ausgeliefert werden. Zu jeder Datei und jedem Ordner soll dabei der Name (mit Verlinkung), das Änderungsdatum und bei Dateien die Größe zu sehen sein.

Vermeidung von clientseitigen Fehlermeldungen

Dass viele typische Funktionen eines Webserver fehlen (siehe Kapitel 4.2.5, Seite 40), sollen Nutzer*innen im Optimalfall nicht merken. Stattdessen soll der Browser mithilfe der korrekten HTTP-Statuscodes und der Header auf die Menge der unterstützten und nicht unterstützten Funktionen aufmerksam gemacht werden, um beispielsweise einen Kompatibilitätsmodus zu aktivieren.

4.2.3 HTTP

Auch ein einfacher Webserver muss grundlegend das Protokoll HTTP unterstützen. Es wird hierbei nicht das Vorhandensein sämtlicher Funktionen gefordert.

HTTP-Methoden

Von den neun HTTP-Methoden (*CONNECT*, *DELETE*, *GET*, *HEAD*, *OPTIONS*, *PATCH*, *POST*, *PUT* und *TRACE*) werden drei implementiert:

GET Die GET-Methode zum Abruf einer Ressource von einem Server ist die wohl wichtigste und meistbenutzte von allen.

HEAD Diese Methode ist neben GET eine der beiden Methoden, die ein Webserver unterstützen muss [63]. Ihr einziger Unterschied zu GET ist, dass in der Antwort der Inhalt der Ressource entfällt, also nur die Header versendet werden. So kann beispielsweise die Größe einer Datei erfragt werden, ohne die Datei selbst vom Server zu laden [64].

OPTIONS Über die OPTIONS-Methode können unter anderem die Methoden erfragt werden, die ein Webserver überhaupt oder für eine bestimmte Ressource unterstützt. Da dies hier nur wenige sind, wird diese Methode implementiert, um auf diesen Umstand hinzuweisen.

HTTP-Header

Es werden nur wenige HTTP-Header bei der Serverantwort gefordert [60]:

Allow (nur für OPTIONS) unterstützte HTTP-Methoden

Connection Soll die Verbindung weiter genutzt werden?

Content-Length Größe der Antwort-Daten (ohne Header) in Bytes. Dieser Header sollte laut RFC¹⁶ immer gesendet werden [65, Sek. 3.3.2].

Content-Type Der auch MIME-Typ genannte Header gibt Auskunft über die Art der gesendeten Daten, beispielsweise `text/html` für HTML-Dateien, `image/png` für PNG-Bilder oder `application/pdf` für PDF-Dokumente

Date Aktuelles Datum und Uhrzeit. Dieser Header ist laut RFC vorgeschrieben [66, Sek. 7.1.1.2].

Server Name des Webserver-Programms mit Versionsnummer

4.2.4 SSL/TLS

Aktuell sind etwa 90 % aller HTTP-Verbindungen verschlüsselt [67]. Die Unterstützung von SSL/TLS ist damit offensichtlich ein wichtiger Bestandteil eines Webservers. Daher soll neben unverschlüsselten Verbindungen der Aufbau einer verschlüsselten Verbindung zum Server mit einem selbstgenerierten Zertifikat möglich sein.

4.2.5 Fehlende Funktionen

Neben den hier genannten Funktionen gibt es auch einige Punkte, die von anderen Webservern unterstützt (und teilweise sogar von RFCs vorausgesetzt) werden, hier aber nicht implementiert wurden.

In den folgenden Abschnitten werden kurz die Gründe dafür dargelegt.

¹⁶ *Request For Comments*, technische Spezifikation, hauptsächlich für das Internet

Serverseitige Programmiersprache

Auf einem Webserver läuft normalerweise eine Skriptsprache wie PHP, die serverseitig Code ausführen kann. Für diesen Server wird aus Zeitgründen darauf verzichtet: Der Aufwand ist erheblich größer als der für den Rest des Projekts, und die Implementierung würde den Gesamtaufwand vermutlich verdoppeln.

HTTP-Methoden

Das Fehlen einer Skriptsprache hat zur Folge, dass die Methoden DELETE, PATCH, POST und PUT überflüssig werden: Sie alle haben das Ziel, eine Zustandsänderung auf dem Server auszulösen und setzen dafür eine serverseitige Programmiersprache voraus. Fehlt diese, funktionieren auch die genannten Methoden nicht oder nicht richtig.

Die CONNECT-Methode ist eher für spezielle Anwendungsgebiete wie Proxy-Server geeignet, TRACE zum Debuggen von Netzwerken. Dies spielt für diese Arbeit beides keine Rolle, daher wird hier auf eine Implementierung verzichtet.

5 Implementierungen des Webservers

In diesem Kapitel sollen die drei entstandenen Implementierungen kurz vorgestellt werden.

5.1 Implementierung in C

Die C-Implementierung erfolgte auf Basis von BSD/POSIX-Internet-Sockets. Es werden POSIX-Threads zur Verwaltung von eingehenden Verbindungen verwendet. Für Semaphore wird die SystemV-Variante benutzt.

Die *openssl*-Bibliothek als einzige Fremdbibliothek wird für SSL-Verbindungen genutzt.

Der genaue Aufbau ist im Folgenden beschrieben.

5.1.1 Struktur

Der C-Server ist zur Verbesserung der Übersichtlichkeit in verschiedene Dateien aufgeteilt (vgl. Abbildung 5.1 auf Seite 44). Alle Dateien werden zu Objekten kompiliert und zu einer einzelnen Executable gelinkt.

Das Hauptprogramm mit der *main()*-Methode befindet sich in der Datei *webserver.c*. Die Konfigurationsdatei ist in *config.c* in mehrere Funktionen zum Einlesen und Ausgeben gekapselt. Die Analyse eines Dateipfades und das Einlesen einer Datei erfolgen in *filehandler.c*. Eine Hilfsfunktion zur Bestimmung des MIME-Typs einer Datei befindet sich in *mimetype.c*, andere in *util.c*. Die Implementierung des Servers selbst erfolgt in *server.c* (wie erfolgt die Kommunikation?) und *http.c* (was wird empfangen/versendet?). *ssl.c* enthält dabei Hilfsfunktionen für SSL. In *simplesema.c* sind Funktionen definiert, die den Umgang mit Semaphoren einfacher machen. Diese Datei stammt aus einem Systemprogrammierung-Praktikum von Prof. Beierlein [68].

5.1.2 Ablauf

Vom Einstiegspunkt aus wird zunächst die Konfigurationsdatei eingelesen und ein Semaphore, ein Signal- und ein Exithandler vorbereitet bzw. festgelegt. Für die zwei in der Konfigurationsdatei festgelegten Ports wird daraufhin jeweils ein Socket angelegt und für den SSL-Socket zusätzlich ein SSL-Kontext erstellt.

Schließlich wird in einer Endlosschleife an beiden Sockets auf eingehende Verbindungen gewartet. Bei deren Eintreffen wird ein neuer Thread erstellt, in welchem die Anfrage des Clients gelesen – je nach Port entweder per SSL oder als Klartext – und geparkt wird. Falls eine Ressource angefragt wurde (HTTP-Methode GET oder HEAD), wird überprüft, ob sie existiert und Lesen möglich ist. Wenn die angefragte Ressource ein Ordner ist, wird eine Auflistung der Dateien in diesem erstellt. Dem Client wird die Datei (mit entsprechendem MIME-Typ), die Auflistung oder im Fehlerfall eine entsprechende Fehlerseite mit passendem HTTP-Statuscode zurückgeliefert.

Beendet wird das Programm normalerweise über das Signal *SIGINT*, welches über einen Signalhandler den Exithandler aufruft, der sich um den ordnungsgemäßen Abbau der genutzten Objekte (Sockets, Threads, SSL-Kontext) kümmert. Über Semaphore wird sichergestellt, dass alle Threads ihre Aufgabe abgeschlossen haben, bevor das Programm verlassen wird.

Auf Fehler im Prozess wird auf verschiedene Weise reagiert: Manche, beispielsweise fehlerhafte Anfragen oder nicht gefundene Dateien, gehören zum Ablauf und werden nicht gesondert (als Fehler) behandelt. Andere Probleme führen hingegen zum Abbau der Verbindung oder zum Stopp des Programms. In diesen Fällen wird auf der Konsole eine aussagekräftige Fehlermeldung ausgegeben.

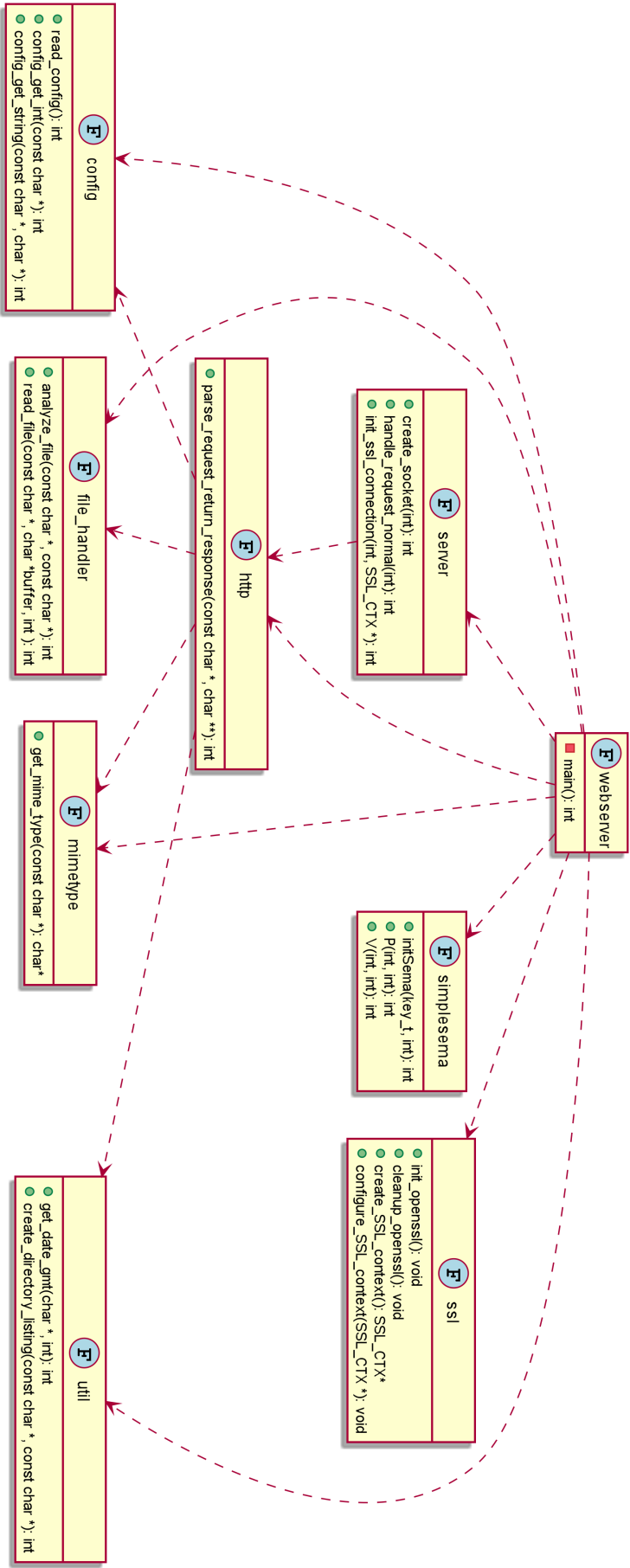


Abbildung 5.1: Dateien C-Server

5.2 Implementierung in Java

Das Java-Programm nutzt (neben den eigenen) ausschließlich interne Java-Pakete, darunter das *java.net*- und *javax.net.ssl*-Paket für den Server, *java.util.concurrent* für diverse Multithreading-Funktionen und *java.io* für Dateioperationen.

5.2.1 Struktur

Das Programm ist in mehrere sog. Pakete (*packages*) unterteilt, welche wiederum verschiedene Klassen enthalten (siehe Abbildung 5.2 auf Seite 47).

Das Paket *config* mit der einzigen Klasse *Config* verwaltet die Konfigurationsdatei und stellt die dort gefundenen Werte anderen Klassen bereit. Die *Mimetype*-Klasse des *files*-Paketes weist wie in C einer Datei den passenden MIME-Typ zu, während *FileUtil* Funktionen zur Validierung von Dateipfaden und Generierung einer Verzeichnisübersicht bietet. Die eigentliche Serverfunktion ist im *server*-Paket realisiert: Hier befinden sich der Einstiegspunkt in das Programm (*Main*) sowie die benötigten Serverklassen *Webserver*, *ServerTask*, *HeaderBuilder* und *SSLUtil*, deren Funktionen in Kapitel 5.2.2 genauer beschrieben sind. Zuletzt existiert noch das *util*-Paket, dessen Klassen *Log* und *Util* einerseits ein abschaltbares Debug-Log und andererseits eine vereinfachende Datums-Funktion bieten.

5.2.2 Ablauf

Gestartet wird das Programm über die *main*-Methode. Dort wird zunächst die Konfigurationsdatei eingelesen und geparkt, um dann für den HTTP- und HTTPS-Port jeweils ein neues *Webserver*-Objekt zu erzeugen. In zwei neuen Threads werden diese beiden gestartet, woraufhin sie jeweils einen neuen Socket anlegen und auf Verbindungen warten.

Sobald eine solche bei einem der Ports ankommt, wird ein *ServerTask* erstellt, der diese bearbeiten soll. Der Task wird einem Threadpool zugewiesen, damit der nächste verfügbare Thread ihn ausführt. Die Anfrage des Clients wird gelesen, geparkt und der angefragte Dateipfad extrahiert. Dieser wird in *files.FileUtil* aufgelöst und auf Existenz und Zugriffsrechte geprüft. Falls es sich bei der angefragten Ressource um einen Ordner handelt, wird auch hier eine Liste der Verzeichnisinhalte erstellt. Aus der Rückgabe wird ein Header mit passendem Statuscode erstellt und ggf. der Dateinhalt angehängt und an den Client gesendet.

Der einzige Unterschied zwischen verschlüsselten und unverschlüsselten Konversationen liegt in der Erstellung des Sockets: Für SSL-Sockets wird in *SSLUtil* zunächst eine *Factory* mit den wichtigen Einstellungen erstellt, welche dann wiederum einen Socket

erzeugt, im zweiten Fall wird nur ein einfacher *ServerSocket* genutzt. Der gesamte restliche Ablauf ist identisch.

Beendet wird das Programm über einen sog. *ShutdownHook*, der zu Beginn eingerichtet wird. Dieser stoppt die beiden *Webserver*, welche danach keine neuen Anfragen mehr annehmen. Es wird dann gewartet, bis alle Tasks des Threadpools abgeschlossen sind. Aufgerufen wird dieser Handler über die Tastenkombination `Strg+C`.

Fehler (als *Exceptions*) werden soweit wie möglich propagiert und behandelt oder explizit ignoriert (meist mit Fehlermeldung).

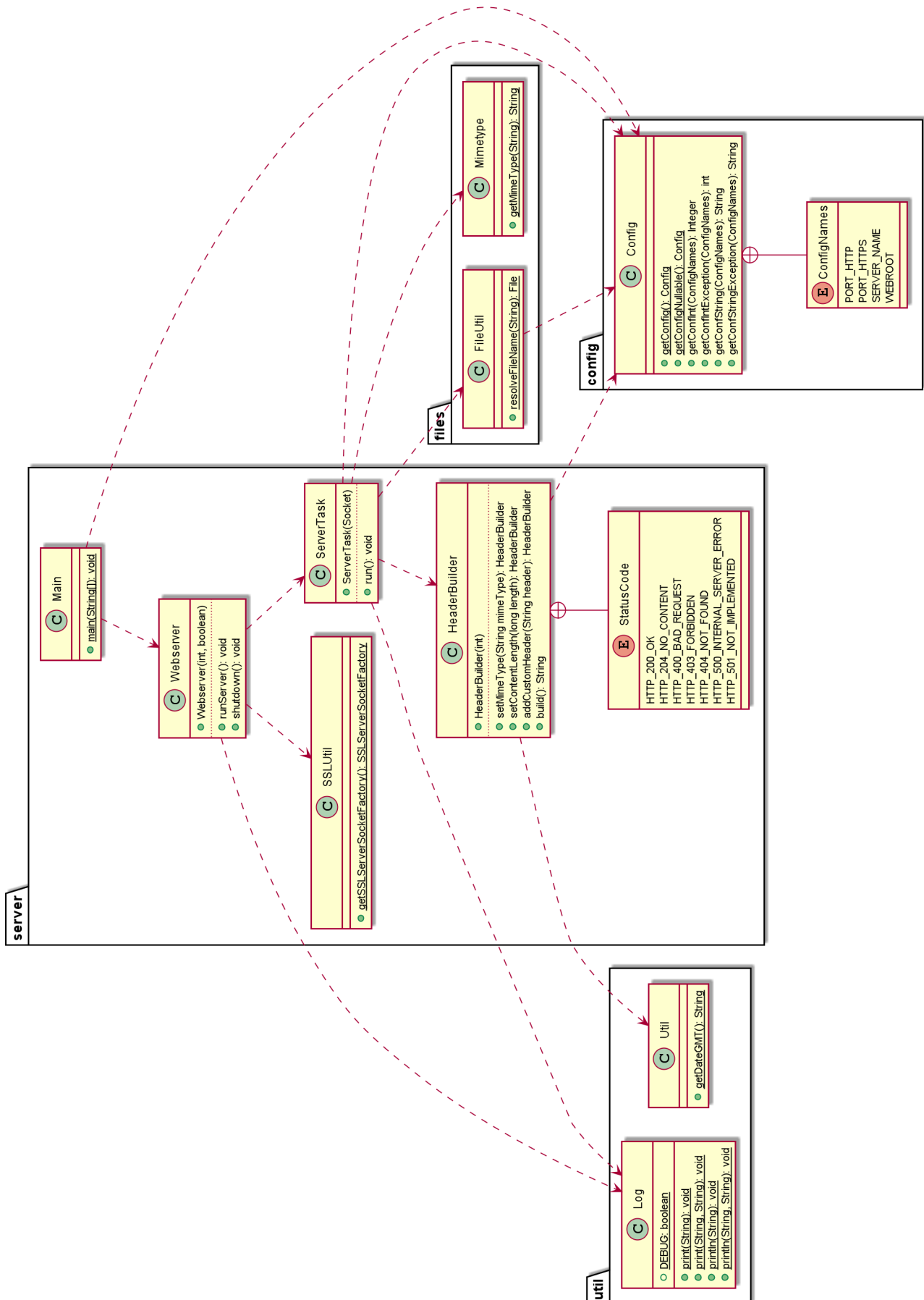


Abbildung 5.2: Klassendiagramm Java

5.3 Implementierung in Rust

Der Rust-Server wurde auf Grundlage des *TcpListener* der Rust-Standardbibliothek (Modul *std::net*) geschaffen. Für SSL-Verbindungen wurde wie in C die *openssl*-Bibliothek genutzt. Die Bindings für Rust ermöglichen es, ohne großen Aufwand einen SSL-Socket (*SslStream<TcpStream>*) zu erstellen und zu nutzen. Multithreading wird mit den Modulen *std::sync* und *std::thread* umgesetzt.

Im Folgenden sind die interne Struktur, der grobe Ablauf sowie die Funktion der genutzten Fremdbibliotheken beschrieben.

5.3.1 Struktur

Zur Verbesserung der Übersichtlichkeit wurde der Code in mehrere Module aufgeteilt: *conf*, *files*, *http*, *server* und *util* (vgl. Abbildung 5.3 auf Seite 50).

conf::config bietet Zugriff auf die Konfigurationsdatei bzw. die ausgelesenen Werte. Die Dateien des *files*-Modules sind zuständig für das Validieren einer angeforderten Datei und das Generieren der Ordneransichten sowie das Finden des MIME-Typs für eine gegebene Datei. *http::header* hilft bei der Erstellung eines HTTP-Headers für die Antwort. In *server::webserver* befindet sich der Code, der die Kommunikation mit dem Client durchführt. Das *util*-Modul enthält Hilfsfunktionen zur Zeitberechnung und -formatierung.

5.3.2 Ablauf

Das Programm startet mit der *main*-Funktion, in der zunächst die Ports für HTTP und HTTPS aus der Konfigurationsdatei abgerufen werden, um dann in zwei neuen Threads jeweils die Webserver-Funktion aufzurufen.

In dieser wird ein Socket (*TcpListener*) angelegt, der auf dem festgelegten Port bis zum Stopp des Programms auf Anfragen wartet. Der SSL-Thread erstellt zusätzlich noch einen *SslAcceptor*, der für die SSL-Verbindungen zuständig ist. Mittels einer generischen Funktion und einem *Trait* wird sichergestellt, dass eine einzelne Handler-Funktion beide Arten von Verbindungen bearbeiten kann.

Für jede neue Verbindung wird ein Thread erstellt, der zunächst die Anfrage des Clients liest und die angeforderte Ressource und HTTP-Methode parst. Die angefragten Dateipfade werden auf Existenz und Leserechte geprüft und dabei normalisiert (*canonicalized*). Falls es sich um einen Ordner handelt, wird eine Übersicht der enthaltenen

Dateien erstellt. Im Fehlerfall wird stattdessen eine Fehlerseite ausgeliefert und ein entsprechender Statuscode gesetzt.

Aus den Metadaten (Größe, MIME-Typ) der Datei wird ein HTTP-Header erstellt, an den der Inhalt der geöffneten Datei ggf. angehängt wird. Diese Daten werden an den Client gesendet.

Gestoppt wird das Programm mit `Strg+C`. Dadurch werden zunächst keine neuen Verbindungen mehr zugelassen, bestehende Übertragungen werden allerdings noch abgeschlossen. Nach Beendigung der beiden Server-Threads (und aller gestarteten Verbindungs-Threads), wird das Programm verlassen.

5.3.3 Fremdbibliotheken

Rust als Sprache fehlt in der Standardbibliothek aktuell noch ein Großteil der Funktionalität, die üblicherweise von Programmiersprachen erwartet wird. Ausgeglichen wird dies durch ein großes Ökosystem von sog. *crates*, die vom Rust-Team selbst oder von Freiwilligen zur Verfügung gestellt werden.

Für den Rust-Server werden mehrere *crates* verwendet, deren Zweck hier kurz beschrieben wird:

ctrlc Diese *crate* stellt einen Handler für die Tastenkombination `Ctrl+C` bereit, über den das Programm beendet werden kann.

chrono Wird genutzt zur Zeitumwandlung.

debug_print Bietet Makros, welche nur in Debug-Builds Text ausgeben.

lazy_static Darüber lassen sich *lazy* (also bei der ersten Benutzung) initialisierte, statische Objekte erschaffen. Normale statische Variablen haben den Nachteil, dass zur Initialisierung nur wenige, konstante (*const*) Funktionen benutzt werden können und so viele Objekte eigentlich nicht statisch möglich sind.

openssl, openssl-sys Ermöglichen die Nutzung der *openssl*-Bibliothek von Rust aus.

regex Damit können reguläre Ausdrücke erstellt und ausgewertet werden.

time (Ursprüngliche) Standard-Zeitbibliothek; wird inzwischen oft durch *chrono* ersetzt.

url, urlencoding Diese beiden *crates* ermöglichen das Dekodieren von URLs.

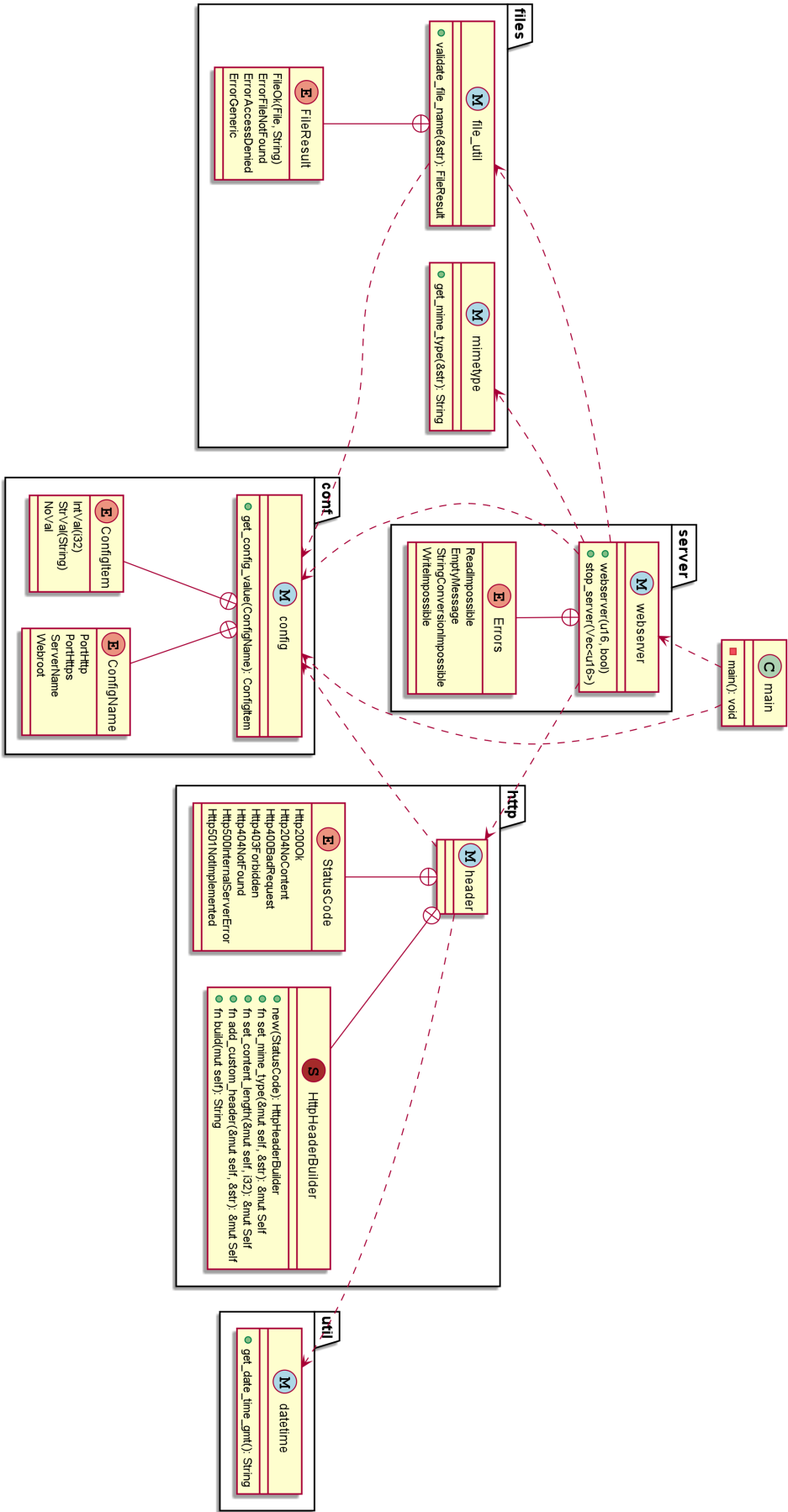


Abbildung 5.3: Moduliagramm Rust

6 Vergleich der Implementierungen

Bei den drei Programmen handelt es sich um verschiedene Implementierungen in verschiedenen Sprachen, die allerdings grundlegend die gleiche Aufgabe erfüllen und das auch größtenteils in der gleichen Weise. Falls ein bestimmtes Problem in einer Programmiersprache eine übliche, effizientere Lösung bietet, wird allerdings diese genutzt.

Es gibt einige Unterschiede zwischen den Implementierungen. Der erste ist die Aufteilung des Codes: In allen drei Sprachen wurde er thematisch getrennt. In C wurden lediglich mehrere Dateien genutzt, welche von anderen Dateien über entsprechende Header eingebunden wurden. Java bietet ein Paketsystem und die Möglichkeit, Klassen aus anderen Paketen (Ordern im Dateisystem) zu importieren und damit für die eigene Klasse verfügbar zu machen. In Rust ist es ähnlich, wobei die Aufteilung hier in Module erfolgt. Ein Modul ist in genau einer Datei festgelegt (die Umkehrung trifft nicht zu: In einer Datei können beliebig viele Module vorhanden sein).

Die anderen Unterschiede betreffen die Funktionalität des Programmes selbst. Die Nutzer*innen bemerken sie allerdings nichts.

Gleichzeitiges Warten auf Verbindungen Damit der Webserver gleichzeitig per SSL/TLS verschlüsselte und unverschlüsselte Verbindungen akzeptieren kann, müssen zwei verschiedene *Sockets* erstellt werden. In C kann per *select* gleichzeitig auf mehrere Sockets gewartet werden. Da das in Java und Rust nicht direkt möglich ist, werden hier stattdessen jeweils zwei *Threads* genutzt.

Erzeugung neuer Threads Für jede neue Verbindung wird ein eigener Thread benötigt. Java bietet die Möglichkeit, einen Thread-Pool zu erstellen und diesem Aufgaben (*Tasks*) zuzuweisen. Threads werden so mehrfach genutzt; der Aufwand für das Erstellen eines Threads fällt nur einmal beim Start des Programmes an. Eine solche Funktionalität existiert in Rust und C nicht; hier muss für jede Verbindung ein neuer Thread erstellt werden.

Diese Punkte können sowohl die Performanz als auch die Komplexität der Programme beeinflussen. Allerdings stehen auch sie für Unterschiede zwischen den Sprachen und werden daher nicht gesondert betrachtet: Wenn sich dadurch beispielsweise die Performanz verbessert, ist dies eben ein Vorteil der Sprache.

6.1 Code-Komplexität

Die Komplexität eines Programmes oder einer Sprache ist ein eher abstraktes Konzept und wird in der Regel subjektiv und im Vergleich mit anderen Programmen bzw. Spra-

chen bestimmt. Es existieren allerdings durchaus verschiedene objektive Metriken, mit denen eine Bewertung durchgeführt werden kann. Trotzdem kann es zwischen ihnen untereinander oder im Gegensatz zu einer subjektiven Bewertung Unterschiede geben, welches Programm das einfachste oder das komplexeste ist.

In der folgenden Sektion werden mehrere dieser Metriken vorgestellt und auf die Programme angewendet, um einen Vergleich zu ermöglichen.

6.1.1 Codezeilen

Um die Komplexität der drei Programmiersprachen beurteilen zu können, ist eine einfache, aber dennoch aussagekräftige Möglichkeit, die Codezeilen (*Source Lines of Code*, *SLOC*) eines Programmes zu zählen. Kommentarzeilen und Leerzeilen werden ignoriert. Diese Zahl kann zwar durch kürzere oder ausführlichere Konstrukte beeinflusst werden. Im vorliegenden Fall ist dieses Problem aber nicht relevant, da alle drei Implementierungen von dem selben Programmierer stammen und damit grundlegend ähnlich programmiert und formatiert sind.

Für die programmierten Webserver ist die Anzahl der Codezeilen in den drei Programmiersprachen in Tabelle 6.1 dargestellt. Sie ist für das C-Programm circa 1,5-mal so groß wie die des Rust-Servers und fast doppelt so groß wie die des Java-Programms, während die beiden letzteren nur etwa 100 Zeilen voneinander abweichen.

Zurückzuführen ist dies wahrscheinlich auf die Notwendigkeit der Fehlerüberprüfungen nach fast jedem Methodenaufruf sowie das Fehlen mancher vereinfachenden Konstrukte in C. Insbesondere das String-Handling in C ist erheblich aufwendiger als in Java und Rust.

Tabelle 6.1: Anzahl der Dateien und Zeilen der Webserver nach Programmiersprache
Gezählt mit *cloc* (<https://github.com/AlDanial/cloc>)

Sprache	Dateien	Leerzeilen	Kommentarzeilen	Codezeilen
<i>C</i>	9	194	226	1227
<i>C/C++ Header</i>	12	31	0	112
<i>Java</i>	11	135	154	717
<i>Rust</i>	12	132	237	807

6.1.2 Halstead-Metrik

Als statisches Analyseverfahren zur Messung der Komplexität kann die Halstead-Metrik genutzt werden, die 1977 von Maurice Halstead vorgestellt wurde [69]. Dabei wird der Programmcode in Operatoren und Operanden unterteilt. Den Operanden werden in der

Regel die Variablen und Konstanten zugeordnet, während alle anderen Bestandteile eines Programmes wie Schlüsselwörter, logische und vergleichende Operatoren oder Funktionen zu den Operatoren gezählt werden. Als Beispiel dient der Kopf der *main*-Funktion eines C-Programmes:

```
int main(int argc, char *argv[])
```

Hier sind `int`, `main`, `()`, `,`, `char`, `*` und `[]` Operatoren und `argc` und `argv` die Operanden [70].¹⁷

Zur Bestimmung der Halstead-Komplexität werden die unterschiedlichen und die insgesamt verwendeten Operatoren und Operanden gezählt. Die Anzahl verschiedener Operatoren (η_1 , im Beispiel 7) und Operanden (η_2 , 2) wird zur Vokabulargröße η summiert, die Gesamtzahl der Operatoren (N_1 , 8) und Operanden (N_2 , 2) bilden zusammen die Implementierungslänge N .

Die Komplexität oder Schwierigkeit D berechnet sich dann mit folgender Formel:

$$D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$$

Das Ergebnis für die drei Sprachen ist in Tabelle 6.2 zu sehen: Hiernach ist die C-Variante immer noch mit einigem Abstand die komplexeste, während der Unterschied zwischen Rust und Java erneut gering ist – diesmal mit leichtem Vorteil für Rust. Die Anzahl verschiedener Operatoren ist in allen drei Sprachen ähnlich, alle anderen Größen führt C mehr oder weniger deutlich an, gefolgt von Rust.

Der Hauptgrund dafür ist allerdings die hohe Gesamtlänge des C-Programms, was zu sehen ist, wenn die Komplexität durch die Anzahl der Zeilen (s. Tabelle 6.1, Seite 52) geteilt wird: Hier liegen die Werte von Java und Rust klar über dem von C.

Tabelle 6.2: Halstead-Metrik der Webserver: Anzahl verschiedener und aller Operatoren und Operanden, absolut und im Verhältnis zur Anzahl der Zeilen

Sprache	η_1	η_2	η	N_1	N_2	N	Komplexität	pro SLOC
C	180	622	802	4758	2211	6969	319.920	0.239
Java	172	454	626	2740	1284	4024	243.225	0.339
Rust	171	532	703	4052	1411	5463	226.768	0.281

¹⁷ Die genaue Einteilung ist nicht festgelegt und unterscheidet sich teilweise von Quelle zu Quelle. Diese wurde in [70] benutzt. [71] zählt Typen wie `int` zu den Operanden, laut [72] werden Funktionsdeklarationen gar nicht gezählt. Da es aber hauptsächlich um den Unterschied zwischen den Programmen geht, ist die exakte Einteilung nicht so wichtig.

6.1.3 Zyklomatische Komplexität

Die zyklomatische Komplexität oder McCabe-Metrik wurde 1976 von Thomas McCabe vorgestellt [73]. Er ging davon aus, dass Verzweigungen und Sprünge in einem Programm die Lesbarkeit verschlechtern und die Komplexität erhöhen. Sie ist daher definiert als die Anzahl möglicher vollständiger Pfade durch ein Programm. Bei einem Programm, das nur aus sequentiellen Anweisungen besteht, ist die zyklomatische Komplexität 1; Bedingungen, Verzweigungen und Schleifen erhöhen sie.

Diese Metrik kann beispielsweise genutzt werden, um die Anzahl der benötigten Testfälle zu bestimmen. Als reine Komplexitätsmetrik, insbesondere im Zusammenhang mit vorgegebenen festen Maximalwerten, wie sie teilweise im Firmenumfeld vorkommen, steht sie in der Kritik, da beispielsweise bei Mehrfachverzweigungen (*switch*) jeder Fall einzeln zählt, obwohl der Code damit unter Umständen lesbarer wird [74]. Trotzdem wird sie hier genutzt, um den Vergleich zwischen den Sprachen zu ziehen.

Die zyklomatische Komplexität für die Programme ist in Tabelle 6.3 abgebildet. Erneut liegt C mit großem Abstand hinten. Für etwa 70 der Pfade ist eine lange Kette aus *ifs* und *e/ses* verantwortlich, die genutzt wurde, weil es in C keine einfachere und effizientere Möglichkeit gibt, einen String mit vielen Werten zu vergleichen und basierend darauf ein Ergebnis zurückzugeben. In Rust und Java konnte an dieser Stelle eine *HashMap* genutzt werden. Aber auch ohne diese 70 Pfade ist der auf diesem Weg ermittelte Komplexitätswert eindeutig: Das Java-Programm besitzt die wenigsten Pfade und C die meisten. Auch im Verhältnis zur Anzahl der Zeilen (s. Tabelle 6.1, Seite 52) ändert sich die Reihenfolge nicht. Ein möglicher Grund sind die vielen Prüfungen der Rückgabewerte auf Fehler in C, welche in Rust und Java teilweise entfallen.

Tabelle 6.3: Zyklomatische Komplexität der Webserver: Anzahl verschiedener Pfade durch das Programm, absolut und im Verhältnis zur Anzahl der Zeilen

Sprache	zyklom. Komplexität	pro SLOC
C	262	0,196
Java	74	0,103
Rust	130	0,161

6.1.4 Kognitive Komplexität

Die Firma *SonarSource*¹⁸ kritisiert die McCabe-Metrik und deren Stellung als De-facto-Standard zur Messung der Komplexität ebenfalls und hat daher ein eigenes Set von Regeln entwickelt, mit denen die sogenannte *Kognitive Komplexität* (*Cognitive Complexity*)

¹⁸ *SonarSource* stellt verschiedene Softwareprodukte, die der Analyse von Programmcode auf Qualität und Sicherheit dienen, her [75].

berechnet werden kann. Das Grundprinzip ist dabei ähnlich der zyklomatischen Komplexität: Jede Unterbrechung im Kontrollfluss erhöht den Komplexitätswert. Allerdings werden Strukturen wie Mehrfachverzweigungen weniger gewichtet, da sie im Allgemeinen den Code eher übersichtlicher machen. Eine *switch*-Anweisung zählt so lediglich einmal statt jedes *case* einzeln. Verschachtelte Verzweigungen werden dagegen höher bewertet: `if (a) { if (b) { ... } }` hat eine Komplexität von 3 (1 für das äußere *if* und 2 für das innere, verschachtelte).

Das Hauptziel dieser Metrik ist es, eine genauere Messung der Verständlichkeit zu bieten, wobei auch modernere Sprachkonstrukte wie *try/catch* oder *Lambda-Ausdrücke*¹⁹ unterstützt werden sollen. Der entstehende Komplexitätswert soll damit (im Vergleich mit anderen Methoden) genauer mit der subjektiven Einschätzung durch Programmierer*innen übereinstimmen. [76]

In Tabelle 6.4 ist die kognitive Komplexität für die drei Programme aufgelistet. Erneut ist der Wert für das C-Programm, sowohl absolut als auch relativ zur Zeilenanzahl, der größte der drei, und Rust liegt zwischen C und Java – allerdings deutlich näher bei Java als bei C. Im Vergleich zur zyklomatischen Komplexität haben sich alle drei Programme verschlechtert, Rust und Java allerdings nur leicht, C deutlich.

Tabelle 6.4: Kognitive Komplexität der Webserver, absolut und im Verhältnis zur Anzahl der Zeilen

Sprache	kognitive Komplexität	pro SLOC
C	324	0,242
Java	91	0,127
Rust	138	0,171

6.1.5 gzip-Kompression

Im Benchmarkgame (vgl. Kap. 3.1) wurde eine weitere Methode angewendet, um die Größe und damit die Komplexität der Programme zu bestimmen: Es wurden zunächst alle Kommentare und Leerzeilen entfernt und anschließend der Code mit *gzip* auf niedrigster Stufe komprimiert (*gzip -1*). Schließlich wurde die Größe der entstehenden Dateien in Bytes gezählt.

Durch die Kompression wird oft wiederholten Zeichenketten ein niedriges Gewicht zugewiesen, da dabei im Optimalfall nur eine Instanz verbleibt und der Rest durch Referenzen ersetzt wird. Eine große Anzahl verschiedener Zeichenketten lässt sich hingegen nicht so gut komprimieren; die komprimierte Version ist ähnlich groß wie die unkomprimierte. Auf diese Weise wird Programmen mit vielen Schlüsselwörtern und

¹⁹ *Lambda-Ausdrücke* sind anonyme Funktionen, also Funktionen ohne Namen

verschiedenen Konstrukten eine höhere Komplexität zugewiesen als Programmen, die beispielsweise nur aus einer sehr oft wiederholten Anweisung bestehen.

Die unkomprimierte und komprimierte Größe der Programme ist in Tabelle 6.5 dargestellt. Die absolute Größe des unkomprimierten C- und des Rust-Programmes sind fast identisch, nur das Java-Programm ist etwa 7 Kilobyte kleiner. Der Komprimierungsvorgang ändert an dieser Reihenfolge nichts. Der Faktor unkomprimiert zu komprimiert ist für alle drei Sprachen ähnlich groß bei etwa 2,7 – Java steht etwas besser da, C und Rust minimal schlechter. Der Abstand zwischen Java und den anderen beiden reduziert sich auf etwa 3 Kilobyte, während der Unterschied zwischen den beiden letzteren nur noch 32 Byte beträgt.

Durch die Kompression ergeben sich somit keine neuen Aussagen über die drei Programme. Allerdings fällt auf, dass das C-Programm trotz deutlich höherer Zeilenzahl gleich groß ist wie der Rust-Server, was auf eine durchschnittlich höhere Zeilenlänge des letzteren hindeutet.

Tabelle 6.5: Größe der bereinigten Programme in Bytes, unkomprimiert und mit *gzip* komprimiert

Befehl: `gzip -1 *`

Sprache	Größe (Bytes, komprimiert)	Größe (Bytes, unkomprimiert)	Faktor
C	10552	28085	2,66
Java	7538	20815	2,76
Rust	10520	28035	2,66

6.2 Performanz

Ein wichtiger Grund für den Einsatz hardwarenaher Sprachen wie C/C++ auch heute noch ist ihre hohe Performanz. Eine mögliche Alternative darf daher nicht wesentlich langsamer sein oder einen wesentlich höheren Speicherverbrauch haben.

In diesem Abschnitt werden daher die Programme in dieser Hinsicht verglichen. Zunächst wird das verwendete Testsystem kurz vorgestellt. Im Anschluss werden die Kompilierzeiten gelistet und gegenübergestellt. Schließlich folgt der eigentliche Vergleich der Performanz: Dabei wird gemessen, wie lange die Beantwortung einer Anfrage bei jeder Version dauert und wie hoch dabei der Ressourcenverbrauch auf dem Serversystem ist.

6.2.1 Testsystem

Sämtliche Performanz-Tests wurden auf einem AMD Ryzen 7 1700 mit acht physischen Kernen und Simultaneous Multithreading (SMT) und 16 Gigabyte Arbeitsspeicher durchgeführt. Das genutzte Betriebssystem war Linux Mint 20 *Ulyana* mit der Kernel-Version 5.4.0-58-generic. *gcc* wurde in Version 9.3.0 (*Ubuntu 9.3.0-17ubuntu1 20.04*) genutzt, *clang* in Version 10.0.0 (*10.0.0-4ubuntu1*) und *rustc* bzw. *cargo* in Version 1.48.0 (*7eac88abb 2020-11-16* bzw. *65cbdd2dc 2020-10-14*). Die Java-Version ist *openjdk 15-ea 2020-09-15*.

Als Client kam ein Laptop mit Intel Core i5-4200M (zwei Kernen und Hyperthreading) sowie 8 Gigabyte RAM zum Einsatz. Das Betriebssystem ist Linux Mint 20.1 *Ulyssa* mit Kernel-Version 5.8.0-38-generic. Das Programm *ApacheBench* war in Version 2.3 (Revision: 1843412) vorhanden.

Die Verbindung zwischen Server und Client erfolgte direkt über ein Ethernet-Kabel. Die maximale Übertragungsbandbreite war 120 MBit/s.

6.2.2 Kompilierung

Im Entwicklungsprozess wird das zu schreibende Programm in vielen Fällen oft neu kompiliert, um es zu testen. Daher sollte dieser Schritt zumindest für unoptimierte Debug-Versionen so schnell gehen wie möglich. Die Zeiten, die für diesen Prozess aufgewendet wurden, wurden für alle Programme mehrfach gemessen, jeweils für die Debug- und die Releaseversion. Das C-Programm wurde sowohl mit *gcc* als auch mit *clang* kompiliert, um Unterschiede zwischen den beiden Compilern feststellen zu können.

Eine vollständige Kompilierung beginnt bei Rust, falls notwendig, mit dem Kompilieren von Abhängigkeiten. Normalerweise kommt dies kaum vor, da die erzeugten Binärdateien in der Regel zwischengespeichert werden. Trotzdem führt dieses Verhalten dazu, dass zumindest der erste Kompiliervorgang vergleichsweise lange dauert. Wenn hingegen nur noch die programmierte *Crate* selbst kompiliert wird, geht es sehr viel schneller.

Trotzdem dauert der Kompilationsprozess bei Rust am längsten (vgl. Abbildung 6.1 und Tabellen B.14 bis B.23 (Seite 83 ff.)). Ein Release-Bau mit Optimierungen²⁰ braucht mehr als drei Sekunden und damit über dreimal so lange wie ein optimierter Bau durch *gcc* oder *clang*²¹, die sich kaum voneinander unterscheiden, und immer noch mehr als

²⁰ `cargo build --release`

²¹ `gcc/clang -O3 -ffast-math -flto`

doppelt so lange wie bei Java. Bei den Debug-Bauten ohne Optimierungen²² holt Rust zwar absolut knapp eine Sekunde auf, relativ gesehen ist der Unterschied allerdings ähnlich groß.

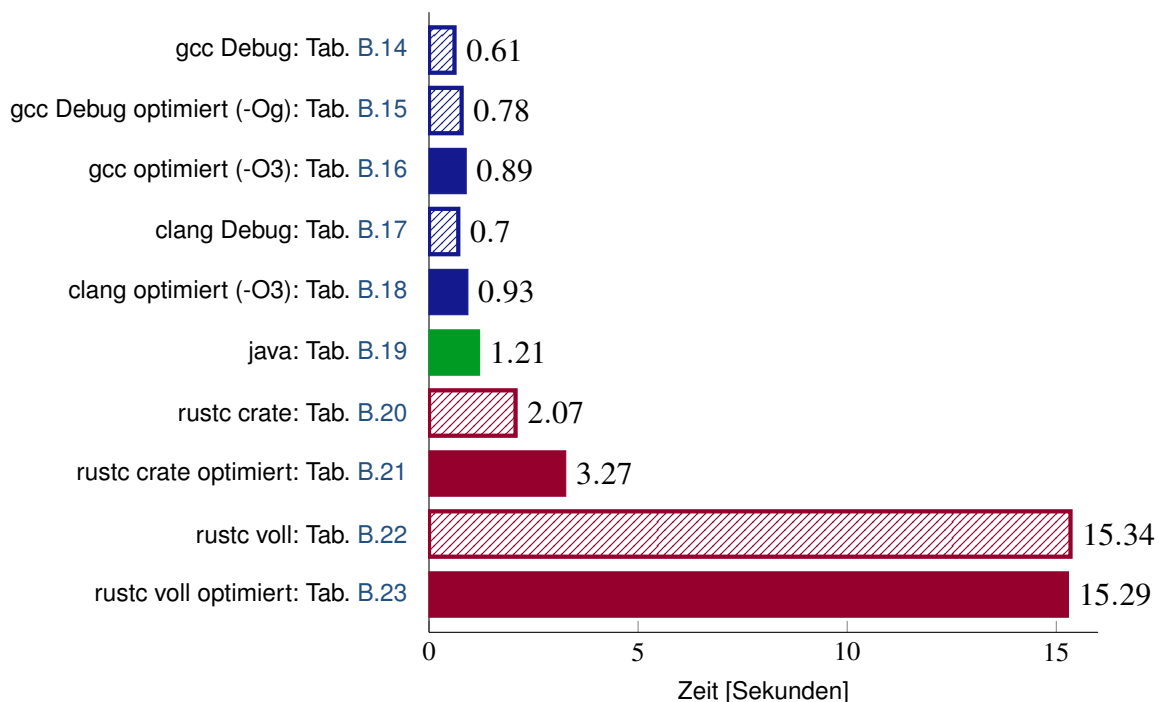


Abbildung 6.1: Kompilationszeiten der Webserver: Durchschnittswerte (Realzeit) aus Tabellen B.14 bis B.23 (S. 83 ff.).

Release-Builds sind gefüllt, Debug-Builds gestreift

6.2.3 Laufzeit-Performanz

Während eine hohe Kompilierzeit während der Entwicklung hinderlich sein kann, ist für Nutzer*innen hauptsächlich das Laufzeitverhalten wichtig. Es gibt praktisch keine Situation, in der ein schnelleres Programm nicht besser wäre.

Daher wird dieser Aspekt auch für die drei Server gemessen.

Der Server wird dabei gestartet, empfängt und beantwortet für 600 Sekunden Anfragen und wird dann wieder beendet. Während der Laufzeit wird alle 0,1 Sekunden die aktuelle CPU- und Speichernutzung ausgelesen. Nach der Beendigung des Prozesses wird außerdem die von der C-Funktion *getrusage* zurückgegebenen Werte gespeichert. Das sind unter anderem die CPU-Zeit, die ein Prozess im *user mode* und *system mode* verbraucht hat und die maximale Speicherauslastung.

²² C: `gcc -g (-Og)` bzw. `clang -g`, Rust: `cargo build`, bei Java gibt es keinen speziellen Debug- oder Release-Bau

Der Client stellt in dieser Zeit mit einer Verzögerung von fünf Sekunden zuerst 2000 Anfragen über HTTP und anschließend weitere 2000 über HTTPS. Dabei werden jeweils 50 Anfragen gleichzeitig gestellt bzw. auf deren Beantwortung gewartet. Die abgerufene Datei ist eine Testdatei, die exakt ein Megabyte (1.048.576 Bytes) groß ist. Dafür wird das Programm *ApacheBench*²³ genutzt. Nach Abschluss des Tests stellt es eine Auswertung bereit, die unter anderem die Anzahl der Anfragen pro Sekunde, die durchschnittliche Zeit bis zur Antwort und die Zeit, in der ein bestimmter Prozentsatz der Anfragen beantwortet wurde, enthält.

Für den Server wird in jedem Fall die optimierte Version verwendet, falls vorhanden. Das C-Programm kommt erneut zweimal vor, einmal kompiliert mit *gcc* und einmal mit *clang*.

Antwortdauer

In Abbildung 6.2 sind die durchschnittlichen Antwortzeiten der Webserver, aufgeteilt auf verschlüsselte und unverschlüsselte Verbindungen, dargestellt. Die Unterschiede zwischen den Versionen sind gering, aber vorhanden.

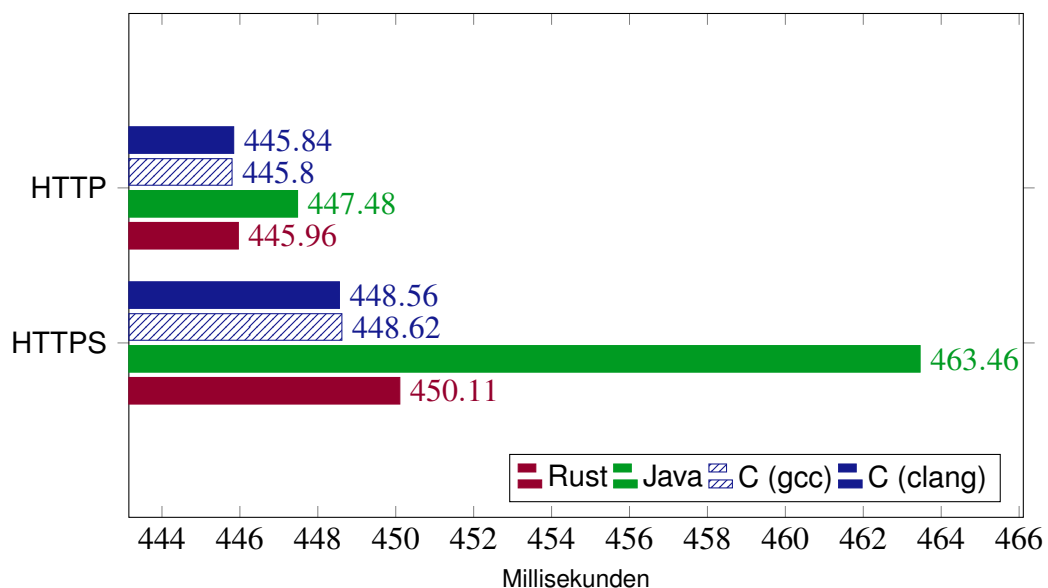


Abbildung 6.2: Durchschnittliche Antwortzeit der Webserver für Anfragen per HTTP und HTTPS

Beide C-Varianten und der Rust-Server beantworten HTTP-Anfragen praktisch in der gleichen Zeit, die Java-Version ist minimal langsamer. Auch bei HTTPS sind die Server in C am schnellsten, dicht gefolgt von Rust. Java benötigt doch einiges länger, im Verhältnis zu C sind es immerhin 15 ms.

²³ <https://httpd.apache.org/docs/current/programs/ab.html>

Das Ergebnis für die Zeiten, in denen ein bestimmter Prozentsatz der Anfragen beantwortet war (Tabelle 6.6), ist bis 90 % ähnlich: Die beiden C-Versionen und das Rust-Programm unterscheiden sich praktisch nicht, Java folgt mit etwa 15 ms Verzögerung. Allerdings zeigt sich, dass sowohl bei C als auch bei Rust die letzten 10 % erheblich länger dauern als bei Java. Besonders groß ist der Unterschied bei HTTP, wo die 2 % der langsamsten Anfragen bei C und Rust mehr als doppelt so lange brauchen wie der Durchschnittswert, während bei Java nur ein geringer Unterschied besteht.

Tabelle 6.6: Antwortzeiten der Webserver in Millisekunden: In welchem Zeitraum kamen x % der Antworten an

	50 %	66 %	75 %	80 %	90 %	95 %	98 %	99 %	100 %
C (clang) HTTP	425	430	433	435	449	652	859	1425	2083
C (gcc) HTTP	430	434	437	439	444	631	792	1388	1862
Java HTTP	444	451	456	460	469	480	493	545	643
Rust HTTP	429	433	435	436	444	561	762	833	1228
C (clang) HTTPS	433	439	442	444	450	594	730	805	1703
C (gcc) HTTPS	431	436	439	441	449	604	777	826	1449
Java HTTPS	446	457	464	470	486	508	724	791	954
Rust HTTPS	437	442	444	447	462	529	715	774	1122

CPU-Nutzung

In Abbildung 6.3 ist die Auslastung der CPU über den Zeitraum des Tests dargestellt. Abgebildet ist der Prozentsatz der Auslastung der einzelnen Kerne, summiert über alle Kerne. Der Maximalwert ist damit 1600 % (100 % $\times$ 16 Kerne).

Klar zu erkennen ist der Wechsel zwischen HTTP- und HTTPS-Anfragen nach etwa 22 Sekunden. Zuvor fordern sowohl die C- als auch der Rust-Server die CPU kaum; die Werte liegen meist im niedrigen zweistelligen Bereich und sind praktisch identisch. Anders hingegen der Java-Server: Gleich zu Beginn steigt die Auslastung auf fast 400 % – hier wird der Bytecode von der JVM geladen –, um dann die HTTP-Anfragen bei etwa 70 % Auslastung zu beantworten.

Für alle vier Varianten ist das Beantworten von verschlüsselten Anfragen deutlich rechenintensiver. Der Java-Server startet diese Phase mit einem Anstieg auf 1550 %, fällt allerdings fast sofort auf etwa 550 % und schwankt dann zwischen 400 und 750 %. Zwischen den anderen drei Versionen besteht erneut kaum ein Unterschied. Sie alle erreichen eine maximale Auslastung von etwa 280 %, danach bewegen sich die Werte zwischen 100 und 250 %.

Nach Beendigung der Anfragen fällt die Auslastung für alle Server – Java und Rust minimal schneller als beide C-Varianten – fast auf 0 %.

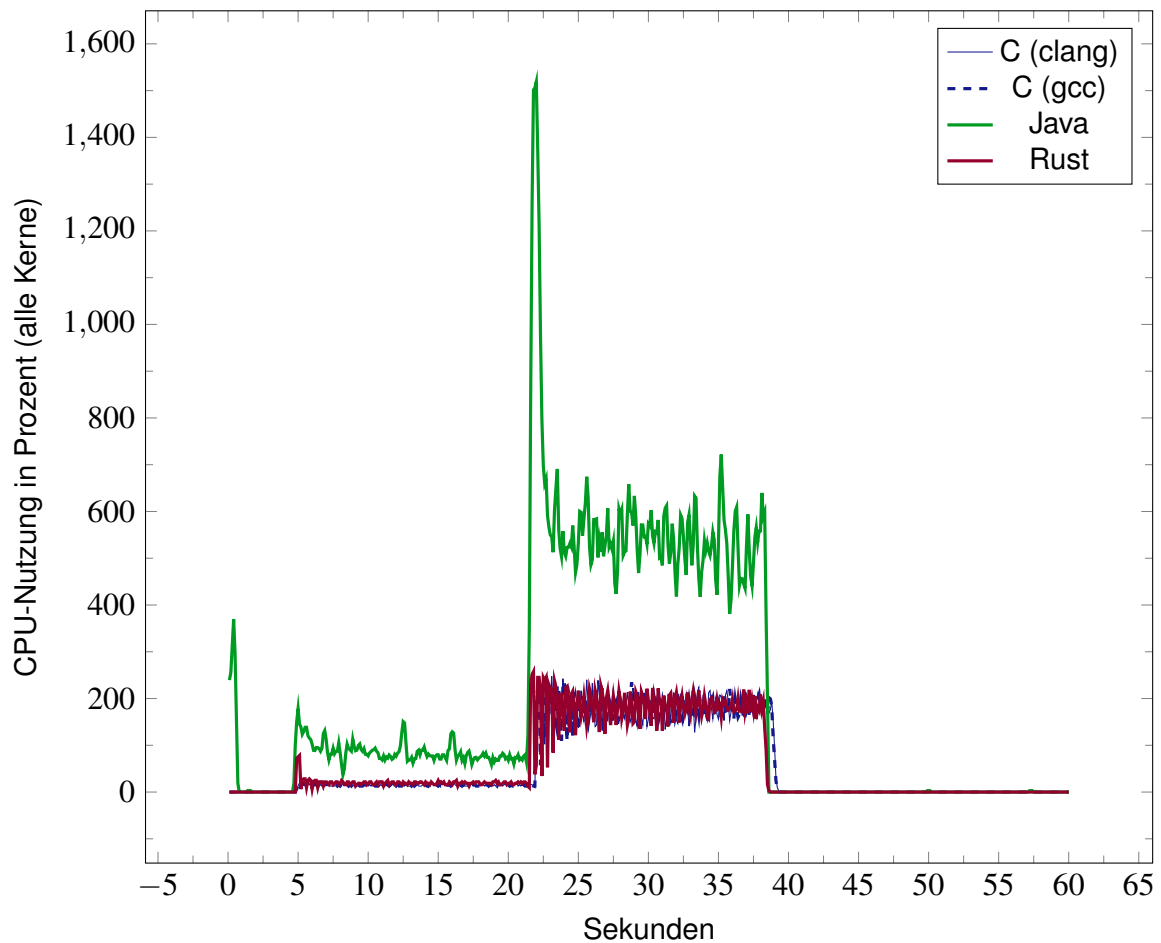


Abbildung 6.3: CPU-Nutzung der Webserver über alle Kerne, in Prozent
Gleitender Mittelwert über 0,2 Sekunden

In Zeiten, in denen für Server pro CPU-Zeit gezahlt wird, ist nicht mehr nur eine schnelle Ausführung einer Berechnung entscheidend, sondern auch die Reduktion der dabei genutzten Rechenzeit.

Daher wird auch diese Zeit für die Programme analysiert (s. Tabelle 6.7). Zu erkennen ist, dass sich die Zeitwerte für beide C-Varianten und Rust kaum unterscheiden, allerdings liegt C leicht vorne. Java nutzt hingegen fast die vierfache Menge der CPU-Zeit im *user mode* und immer noch die dreifache im *system mode*.

Tabelle 6.7: Ressourcennutzung der Webserver über die gesamte Laufzeit: Genutzte CPU-Zeit im *user mode* und *system mode* (Sekunden) und maximale Speichernutzung (*resident set size*, RSS)

Rückgabe der Funktion *getrusage*

	CPU-Zeit <i>user mode</i>	CPU-Zeit <i>system mode</i>	RSS in Megabyte
C (clang)	29,024	6,071	62,32
C (gcc)	28,791	6,560	62,56
Java	107,630	18,147	446,18
Rust	30,160	6,421	116,66

Speichernutzung

Die Speichernutzung während des Tests ist in Abbildung 6.4 zu sehen. Der Werte stehen für die *Unique Set Size*, was dem aktuell garantiert für einen Prozess reservierten Bereich des Arbeitsspeichers entspricht. Damit ist die Menge an Speicher bestimmt, die freigegeben würde, wenn der Prozess zu dem Zeitpunkt beendet würde. Dieses Kriterium wird als repräsentativster Wert für die aktuelle Speichernutzung eines Programmes angesehen.

Rust- und C-Server benutzen zu Beginn fast keinen Speicher, während der Java-Server auch ohne Anfragen bereits mehr als 80 Megabyte benötigt.

Bei Beginn der Anfragen steigt dieser Wert für Java zunächst schnell auf 150 Megabyte und dann bis zum Ende der Phase nach einem zufällig erscheinenden Muster bis auf 230 Megabyte an. Rust steigt ebenso schnell auf ca. 120 Megabyte und behält diesen Wert dann auch bei. Bei beiden C-Versionen erhöht sich der Wert etwas langsamer und nur auf etwa 60 Megabyte, wo er sich ebenfalls nicht mehr verändert.

Zu Beginn der HTTPS-Phase steigert sich der Speicherverbrauch bei Rust und C leicht um wenige Megabyte, um sich dann den Rest dieses Zeitraums nicht mehr zu bewegen. Java hingegen braucht fast sofort 330 Megabyte, danach schwankt der Wert zwischen 280 und 380 Megabyte.

Nach Beendigung der Anfragen passiert kaum etwas. Keines der Programme gibt größere Mengen Speicher frei, auch wenn man bei Rust und C einen leichten Knick nach unten sieht. Die Java-Version pendelt sich hingegen bei etwa 330 Megabyte ein.

Wie in Tabelle 6.7 zu sehen ist, liegt der Maximalwert insgesamt für C bei etwa 62 Megabyte, Rust folgt auf dem zweiten Platz mit 117 Megabyte und Java mit großem Abstand mit 446 Megabyte.

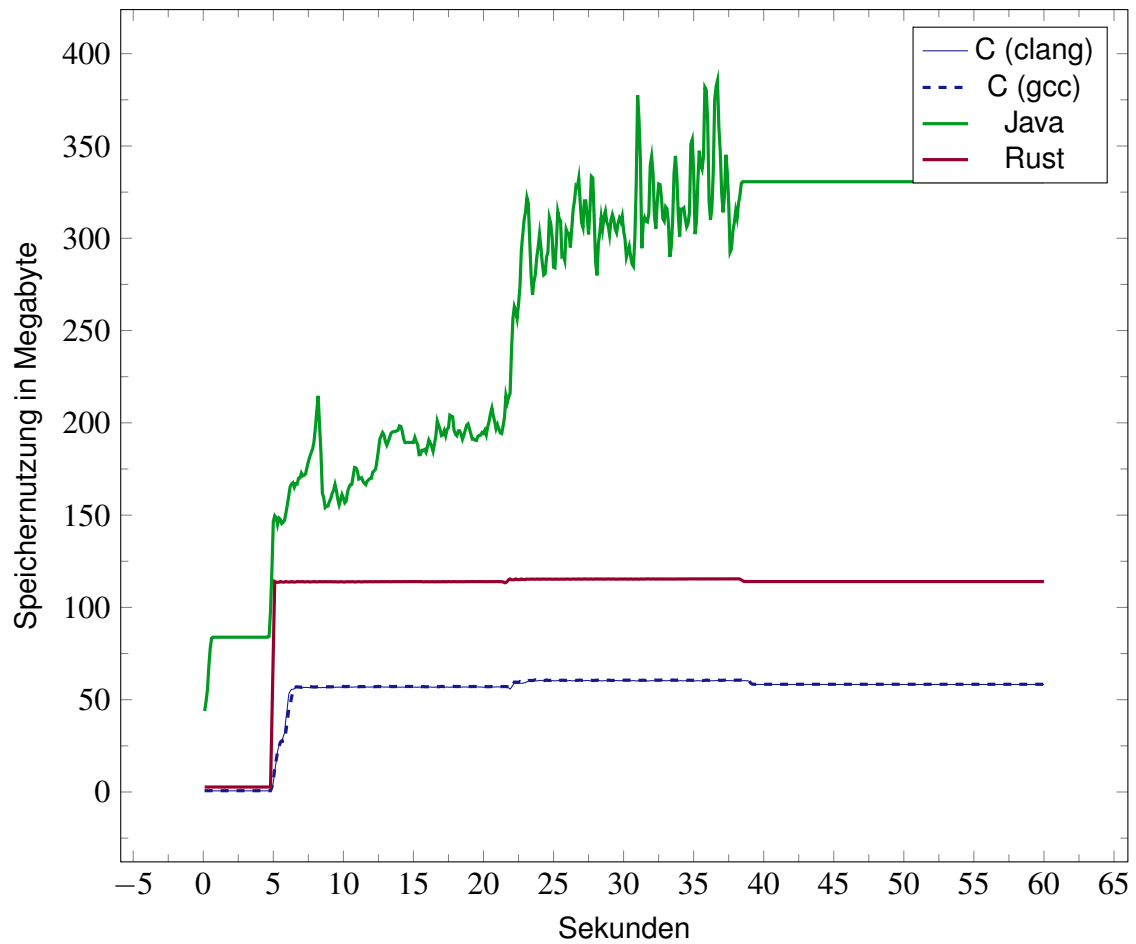


Abbildung 6.4: Speichernutzung der Webserver, in Megabyte
Gleitender Mittelwert über 0,2 Sekunden

7 Fazit

Es hat sich gezeigt, dass Rust zwar die Designziele erfüllt, aber trotzdem nicht in allen Fällen als Programmiersprache geeignet ist.

7.1 Rust als Ersatz für hardwarenahe Sprachen wie C

Der Vergleich zwischen C und Rust war relativ eindeutig: In fast jedem betrachteten Fall erwies sich Rust als besser oder mindestens ähnlich gut.

Das Hauptziel, sicherer zu sein als C, wird (unter normalen Umständen²⁴) definitiv erreicht. Die typischen Gründe für Sicherheitslücken in C werden in Rust großteils verhindert, und das in vielen Fällen schon beim Kompilieren. Andere lassen sich prinzipiell nicht beheben: Logikfehler und *race conditions* können nur die Programmierer*innen selbst vermeiden.

Aber auch in der Praxis zeigt sich: Viele Sicherheitslücken der letzten Jahre wären vielleicht verhindert worden, wenn das entsprechende Programm in Rust geschrieben worden wäre.

Dies liegt vermutlich daran, dass Rust genau aus diesem Grund entwickelt wurde und die Erfahrungen aus fast 40 Jahren C einfließen konnten. Ein anderes Ergebnis wäre regelrecht überraschend gewesen.

Aber Rust soll nicht nur sicherer sein als C, sondern auch mindestens genauso schnell. Und auch dieses Ziel wird erfüllt. Laut den Benchmarks ist Rust sogar schneller als die C-Varianten und wird nur von C++ geschlagen. Für das Beispielprogramm ist das Ergebnis anders, hier ist die C-Variante schneller, allerdings nur minimal.

Beim dritten untersuchten Aspekt, der Komplexität, ist das Ergebnis nicht eindeutig. Der prototypische C-Webserver ist zwar in fast jeder Metrik komplexer als die Rust-Variante, bei den Benchmarks hingegen ist das Ergebnis umgekehrt. Dieser Unterschied könnte auf die verschiedenen Längen der beiden Programme zurückzuführen sein – die Benchmarks sind etwa 100 Zeilen lang, die Webserver 1000 –, auf die anderen Programmarten oder auf die verwendeten Metriken. Ein weiterer Vergleich mit längeren oder kürzeren Programmen könnte genauere Ergebnisse liefern.

²⁴ Mittels *unsafe*-Blöcken können auch in Rust Sicherheitslücken verursacht werden. Allerdings ist das Fehlen der entsprechenden Prüfungen im Gegensatz zu C nicht der Standard.

Zusammenfassend lässt sich sagen: Es gibt kaum noch einen Grund, C weiter zu verwenden – insbesondere für neue Programme. Die Sicherheit zu vernachlässigen, um eine höhere Geschwindigkeit zu erzielen, lohnt sich generell nur in wenigen Fällen. Und Rust sorgt dafür, dass diese Entscheidung nicht mehr getroffen werden muss.

Eine Rust ebenbürtige Alternative ist vielleicht C++. Mit Konzepten wie *RAII* oder *Smart Pointern* werden ebenfalls viele Sicherheitsprobleme von C vermieden, und die Geschwindigkeit von C++-Programmen ist laut den Benchmarks sogar noch höher.

7.2 Rust als Ersatz für Hochsprachen wie Java

Die Gründe für die Entwicklung mit Java sind andere als für die mit C oder Rust. Hier geht es eher darum, einfachen und sicheren Code zu schreiben, selbst wenn das zu Lasten der Performanz passiert.

Obwohl Rust auch in dieser Hinsicht vieles besser macht als C: Die Komplexität des Java-Servers ist in allen Metriken die niedrigste. Die Werte für die Benchmarks zeigen erneut ein anderes Ergebnis; die möglichen Gründe sind die gleichen wie in Kapitel 7.1.

Beim Thema Sicherheit unterscheiden sich Java und Rust kaum. Schwerwiegende Sicherheitslücken sind in Programmen in beiden Sprachen eigentlich nicht möglich. In Rust führen Fehler normalerweise zu einer *panic*, in Java zu einer *Exception*, aber das Ergebnis ist in der Regel gleich: Das Programm (oder zumindest der Thread) wird mit einer Fehlermeldung beendet. Java bietet allerdings die Möglichkeit, *Exceptions* abzufangen und zu behandeln, was für eine *panic* in Rust nicht möglich ist²⁵.

Den Vergleich der Geschwindigkeit entscheidet Rust klar für sich, sowohl in den Benchmarks als auch bei den Webservern. Dieser Punkt zeigt deutlich, warum auch heute noch viele Programme in C/C++ geschrieben werden: Trotz vieler Verbesserungen der JVM kann Java nicht mit diesen Sprachen mithalten.

Damit müssen Entwickler*innen sich bei der Wahl der Programmiersprache entscheiden, ob ein einfaches, langsames oder ein kompliziertes, schnelles Programm gewünscht ist. In vielen Fällen ist die Performanz nicht so wichtig: Gerade bei Anwendungen für Endnutzer ohne größere Hintergrundberechnungen sind praktisch alle Programmiersprachen so schnell, dass Nutzer*innen keine Verzögerung bemerken. Und falls es Berechnungen dieser Art gibt, ist es sogar möglich, Rust über das JNI anzusprechen [77] und sie damit auszulagern.

²⁵ Es gibt eine Funktion (*catch_unwind*), die das kann, aber diese sollte hauptsächlich für Nicht-Rust-Code verwendet werden und nicht als Ersatz für *try/catch* [33, Funktion *std::panic::catch_unwind*].

Literatur

- [1] Wikipedia, Hrsg., *List of programming languages*, 2021. Adresse: https://en.wikipedia.org/w/index.php?title=List_of_programming_languages&oldid=999721196 (besucht am 13.01.2021).
- [2] StackOverflow, *Stack Overflow Insights - Developer Hiring, Marketing, and User Research*, Jan. 2021. Adresse: <https://insights.stackoverflow.com/survey> (besucht am 13.01.2021).
- [3] StackOverflow, Hrsg., *Stack Overflow Developer Survey 2020*, Dez. 2020. Adresse: <https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved> (besucht am 18.12.2020).
- [4] —, *Stack Overflow Developer Survey 2015*, Jan. 2021. Adresse: <https://insights.stackoverflow.com/survey/2015#tech-super> (besucht am 13.01.2021).
- [5] P. Walton, *C++ design goals in the context of Rust*, 2010. Adresse: <https://pcwalton.blogspot.com/2010/12/c-design-goals-in-context-of-rust.html> (besucht am 13.01.2021).
- [6] Rust, Hrsg., *Rust Programming Language*, Jan. 2021. Adresse: <https://www.rust-lang.org/> (besucht am 14.01.2021).
- [7] P. A. Henning und H. Vogelsang, Hrsg., *Handbuch Programmiersprachen: Softwareentwicklung zum Lernen und Nachschlagen : [ABAP, Ada, C, C#, C++, Delphi, Eiffel, FORTRAN, Java, JavaScript, LISP, Maple, Mathematica, MATLAB, Perl, PHP, PROLOG, Python, Ruby, Smalltalk, SQL, Tcl/Tk, Visual Basic.NET]*. München: Hanser, 2007, ISBN: 978-3-4464-0558-5. Adresse: http://deposit.dnb.de/cgi-bin/dokserv?id=2802801&prov=M&dok_var=1&dok_ext=htm.
- [8] R. W. Sebesta, *Concepts of programming languages*, 8., [internat.] ed. Boston, Mass.: Pearson Addison-Wesley, 2008, ISBN: 978-0-3215-0968-0.
- [9] J. Hohmann, *Der Plankalkül im Vergleich mit algorithmischen Sprachen*, Ser. (Informatik und Operations-research. Darmstadt: Toeche-Mittler, 1979, Bd. 7), ISBN: 3-878-20028-5.
- [10] W. Bolkart, *Programmiersprachen der vierten und fünften Generation*, Ser. McGraw-Hill-Texte. Hamburg: McGraw-Hill, 1987, ISBN: 3-890-28102-8.
- [11] T. Pratt und M. V. Zelkowitz, *Programmiersprachen: Design und Implementierung*. München: Prentice-Hall, 1998, ISBN: 3-827-29547-5.
- [12] PCMAG, Hrsg., *Definition of fourth-generation language*. Adresse: <https://www.pcmag.com/encyclopedia/term/fourth-generation-language> (besucht am 19.01.2021).

- [13] P. Howard, „What makes a 5GL? Impossible, but ...“, *The Register*, Jan. 2007. Adresse: <https://www.theregister.com/2007/01/17/5gl/> (besucht am 19.01.2021).
- [14] GeeksforGeeks, Hrsg., *Generation of Programming Languages - GeeksforGeeks*, 2017. Adresse: <https://www.geeksforgeeks.org/generation-programming-languages/> (besucht am 19.01.2021).
- [15] TIOBE, Hrsg., *TIOBE Index for January 2021: Python is TIOBE's Programming Language of 2020!* Jan. 2021. Adresse: <https://www.tiobe.com/tiobe-index/> (besucht am 23.01.2021).
- [16] B. W. Kernighan und D. M. Ritchie, *The C programming language*, 2. ed., Ser. Prentice-Hall software series. Englewood Cliffs, NJ: Prentice-Hall, 1988, ISBN: 0-13-110370-9.
- [17] D. M. Ritchie, *The Development of the C Language*, Association for Computing Machinery, Inc., Hrsg., 1993. Adresse: <https://csapp.cs.cmu.edu/3e/docs/chistory.html> (besucht am 04.12.2020).
- [18] M. Bautsch, „Cycles of Software Crises: How to avoid insecure and uneconomic software“, *ENISA Quarterly*, Jg. 3, Nr. 4, S. 3–5, 2007. Adresse: <https://www.enisa.europa.eu/publications/eqr-archive/issues/eqr-q4-2007-vol.-3-no.-4> (besucht am 06.12.2020).
- [19] J. Byous, *Java Technology: The Early Years*, Sun Microsystems, Hrsg., 2003. Adresse: <https://web.archive.org/web/20080530073139/http://java.sun.com/features/1998/05/birthday.html> (besucht am 22.12.2020).
- [20] K. Southwick, *High noon: The inside story of Scott McNealy and the rise of Sun Microsystems*. New York: Wiley, 1999, ISBN: 978-0-4712-9713-0.
- [21] C. Ullenboom, *Java ist auch eine Insel: Einführung, Ausbildung, Praxis*, 15., aktualisierte und überarbeitete Auflage, Ser. Rheinwerk Computing. 2020, ISBN: 978-3-8362-7737-2.
- [22] D. Topic, *Moving to a Plugin-Free Web | Oracle Java Platform Group, Product Management Blog*, Oracle, Hrsg., Jan. 2016. Adresse: <https://blogs.oracle.com/java-platform-group/moving-to-a-plugin-free-web> (besucht am 08.01.2021).
- [23] R. Gunkar, *What is the Role of Java in the IT Industry? - GeeksforGeeks*, GeeksforGeeks, Hrsg., 2019. Adresse: <https://www.geeksforgeeks.org/what-is-the-role-of-java-in-the-it-industry/> (besucht am 26.12.2020).
- [24] G. Hoare, *Rust prehistory*, GitHub, 2009. Adresse: <https://github.com/graydon/rust-prehistory/blob/master/README.md> (besucht am 17.12.2020).
- [25] —, *[rust-dev] stage1/rustc builds: Rust-dev mailing list*, Mozilla, Hrsg., Apr. 2011. Adresse: <https://mail.mozilla.org/pipermail/rust-dev/2011-April/000330.html> (besucht am 17.12.2020).

- [26] B. Eich, *Future Tense*, Mozilla, Hrsg., Apr. 2011. Adresse: <https://www.slide share.net/BrendanEich/future-tense-7782010> (besucht am 17. 12. 2020).
- [27] S. Yegulalp, *Mozilla binds Firefox's fate to the Rust language*, InfoWorld, Hrsg., 2017. Adresse: <https://www.infoworld.com/article/3165424/mozilla-binds-firefoxs-fate-to-the-rust-language.html> (besucht am 20. 01. 2021).
- [28] The Rust Core Team, Hrsg., *Announcing Rust 1.0 | Rust Blog*, Mai 2015. Adresse: <https://blog.rust-lang.org/2015/05/15/Rust-1.0.html> (besucht am 17. 12. 2020).
- [29] B. Anderson, *The Rust Compilation Model Calamity*, PingCAP, Hrsg., Jan. 2020. Adresse: <https://pingcap.com/blog/rust-compilation-model-calamity> (besucht am 15. 12. 2020).
- [30] S. Klabnik und C. Nichols, *The Rust Programming Language*. 2019, ISBN: 978-1-7185-0044-0. Adresse: <https://doc.rust-lang.org/stable/book/>.
- [31] Rust, Hrsg., *The Cargo Book*. Adresse: <https://doc.rust-lang.org/cargo/index.html> (besucht am 12. 01. 2021).
- [32] Rust, Hrsg., *Rust By Example*, Dez. 2020. Adresse: <https://doc.rust-lang.org/rust-by-example/> (besucht am 13. 01. 2021).
- [33] Rust, Hrsg., *The Rust Standard Library: Documentation*. Adresse: <https://doc.rust-lang.org/std/index.html> (besucht am 18. 01. 2021).
- [34] Rust, Hrsg., *Guide to Rustc Development*. Adresse: <https://rustc-dev-guide.rust-lang.org/> (besucht am 12. 01. 2021).
- [35] A. Turon, *Abstraction without overhead: traits in Rust | Rust Blog*, Rust Blog, Hrsg., Nov. 2015. Adresse: <https://blog.rust-lang.org/2015/05/11/traits.html> (besucht am 12. 01. 2021).
- [36] *Which programming language is fastest? | Computer Language Benchmarks Game*, Dez. 2020. Adresse: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html> (besucht am 10. 01. 2021).
- [37] *Toy benchmark task descriptions | Computer Language Benchmarks Game*, Dez. 2020. Adresse: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/summary.html> (besucht am 11. 01. 2021).
- [38] The Rust Survey Team, Hrsg., *Rust Survey 2019 Results | Rust Blog*, Apr. 2020. Adresse: <https://blog.rust-lang.org/2020/04/17/Rust-survey-2019.html> (besucht am 15. 12. 2020).
- [39] Rust, Hrsg., *Frequently Asked Questions · The Rust Programming Language*, Dez. 2018. Adresse: <https://prev.rust-lang.org/en-US/faq.html> (besucht am 15. 12. 2020).

- [40] The Rust Survey Team, Hrsg., *Rust Survey 2020 Results | Rust Blog*, Dez. 2020. Adresse: <https://blog.rust-lang.org/2020/12/16/rust-survey-2020.html> (besucht am 17.12.2020).
- [41] *Which benchmark programs are fastest? | Computer Language Benchmarks Game*, Dez. 2020. Adresse: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/which-programs-are-fastest.html> (besucht am 10.01.2021).
- [42] R. Aigner, M. Kofler und A. Zingsheim, *Hacking & Security: Das umfassende Handbuch*, 2., aktualisierte und erweiterte Auflage. 2020, ISBN: 978-3-8362-7191-2.
- [43] J. Erickson, *Hacking: Die Kunst des Exploits*, Deutsche Ausgabe der 2. amerikanischen Auflage 2009, 8., korrigierter Nachdruck. Heidelberg: dpunkt.verlag, 2009, ISBN: 978-3-8986-4536-2.
- [44] scut, *Exploiting Format String Vulnerabilities*, TESO, Hrsg., Sep. 2001. Adresse: <https://cs155.stanford.edu/papers/formatstring-1.2.pdf> (besucht am 17.01.2021).
- [45] Rust, Hrsg., *The Rustonomicon: The Dark Arts of Unsafe Rust*. Adresse: <https://doc.rust-lang.org/nomicon/index.html> (besucht am 18.01.2021).
- [46] OWASP, Hrsg., *Memory leak*. Adresse: https://owasp.org/www-community/vulnerabilities/Memory_leak (besucht am 18.01.2021).
- [47] Mitre, Hrsg., *CWE-401: Missing Release of Memory after Effective Lifetime (4.3): Common Weakness Enumeration*, Aug. 2020. Adresse: <https://cwe.mitre.org/data/definitions/401.html> (besucht am 18.01.2021).
- [48] OWASP, Hrsg., *Doubly freeing memory*. Adresse: https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory (besucht am 18.01.2021).
- [49] Mitre, Hrsg., *CWE-415: Double Free (4.3): Common Weakness Enumeration*, Dez. 2020. Adresse: <https://cwe.mitre.org/data/definitions/415.html> (besucht am 18.01.2021).
- [50] OWASP, Hrsg., *Using freed memory*. Adresse: https://owasp.org/www-community/vulnerabilities/Using_freed_memory (besucht am 18.01.2021).
- [51] Mitre, Hrsg., *CWE-416: Use After Free (4.3): Common Weakness Enumeration*, Dez. 2020. Adresse: <https://cwe.mitre.org/data/definitions/416.html> (besucht am 18.01.2021).
- [52] NIST, Hrsg., *CVE-2002-0656: National Vulnerability Database*, 2002. Adresse: <https://nvd.nist.gov/vuln/detail/CVE-2002-0656> (besucht am 21.01.2021).
- [53] A. Sotirov, *Apache OpenSSL heap overflow exploit*, 2002. Adresse: <http://www.phreedom.org/research/exploits/apache-openssl/> (besucht am 20.01.2021).

- [54] J. Gillies und R. Cailliau, *How the Web was born: The story of the World Wide Web*. Oxford: Oxford University Press, 2000, ISBN: 0-192-86207-3. Adresse: <http://www.loc.gov/catdir/enhancements/fy0637/2001274436-d.html>.
- [55] CERN, Hrsg., *A short history of the Web | CERN*, Dez. 2020. Adresse: <https://home.cern/science/computing/birth-web/short-history-web> (besucht am 10. 12. 2020).
- [56] CERN, Hrsg., *Status of the CERN httpd*, Mai 2018. Adresse: <https://www.w3.org/Daemon/> (besucht am 10. 12. 2020).
- [57] W3Techs, Hrsg., *Usage Statistics and Market Share of Web Servers, January 2021*, Jan. 2021. Adresse: https://w3techs.com/technologies/overview/web_server (besucht am 10. 01. 2021).
- [58] Netcraft Ltd, Hrsg., *November 2020 Web Server Survey*, Nov. 2020. Adresse: <https://news.netcraft.com/archives/2020/11/30/november-2020-web-server-survey.html> (besucht am 10. 12. 2020).
- [59] B. Laurie, P. Laurie und P. Klicman, *Apache: Das umfassende Handbuch ; [behandelt Apache 2.0 und 1.3, 2. Aufl., dt. Ausg. der 3. Aufl.* Beijing: O'Reilly, 2003, ISBN: 3-897-21356-7.
- [60] MDN Web Docs, *HTTP request methods*, Mozilla, Hrsg., Dez. 2020. Adresse: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods> (besucht am 10. 12. 2020).
- [61] I. Münch, *Apache Webserver Sicherheitsstudie*, Bundesamt für Sicherheit in der Informationstechnik, Hrsg., 2002. Adresse: https://web.archive.org/web/20130407212843/https://www.bsi.bund.de/cae/servlet/contentblob/477104/publicationFile/30664/Apache_2003_pdf.pdf (besucht am 10. 01. 2021).
- [62] W3Techs, Hrsg., *Usage Statistics and Market Share of Server-side Programming Languages for Websites, December 2020*, Dez. 2020. Adresse: https://w3techs.com/technologies/overview/programming_language (besucht am 12. 12. 2020).
- [63] MDN Web Docs, *501 Not Implemented*, Mozilla, Hrsg., Aug. 2020. Adresse: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/501> (besucht am 02. 11. 2020).
- [64] —, *HEAD*, Mozilla, Hrsg., Aug. 2020. Adresse: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD> (besucht am 02. 11. 2020).
- [65] Internet Engineering Task Force, Hrsg., *RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, 2014. Adresse: <https://tools.ietf.org/html/rfc7230> (besucht am 16. 01. 2021).
- [66] Internet Engineering Task Force, Hrsg., *RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, 2014. Adresse: <https://tools.ietf.org/html/rfc7231> (besucht am 16. 01. 2021).

-
- [67] Google, Hrsg., *HTTPS-Verschlüsselung im Web – Google Transparenzbericht*, Jan. 2021. Adresse: <https://transparencyreport.google.com/https/overview> (besucht am 23. 01. 2021).
 - [68] T. Beierlein, *Systemprogrammierung: Beispiele zur Nutzung von Shared Memory und Semaphoren*. (besucht am 15. 12. 2020).
 - [69] M. H. Halstead, *Elements of software science*, Ser. Operating and programming systems series. New York: North Holland, 1977, Bd. 2, ISBN: 0-444-00205-7.
 - [70] Wikipedia, Hrsg., *Halstead complexity measures*, 2020. Adresse: https://en.wikipedia.org/w/index.php?title=Halstead_complexity_measures&oldid=993628212 (besucht am 05. 01. 2021).
 - [71] X.-N. Cullmann und K. Lambertz, „Software: Komplexität und Qualität von Software,“ *MSCoder*, Jg. 2, Nr. 1, S. 36–43, 2007. Adresse: https://www.verifysoft.com/de/cmtp_mscoder.pdf (besucht am 05. 01. 2021).
 - [72] S. Virmani, *Software Engineering | Halstead's Software Metrics - GeeksforGeeks*, GeeksforGeeks, Hrsg., 2017. Adresse: <https://www.geeksforgeeks.org/software-engineering-halsteads-software-metrics/> (besucht am 05. 01. 2021).
 - [73] T. J. McCabe, „A Complexity Measure,“ *IEEE Transactions on Software Engineering*, Jg. SE-2, Nr. 4, S. 308–320, 1976. Adresse: <http://www.literateprogramming.com/mccabe.pdf> (besucht am 05. 01. 2021).
 - [74] B. Hummel, *McCabe's Cyclomatic Complexity and Why We Don't Use It*, CQSE, Hrsg., Mai 2014. Adresse: <https://www.cqse.eu/en/news/blog/mccabe-cyclomatic-complexity/> (besucht am 05. 01. 2021).
 - [75] SonarSource, Hrsg., *Code Quality and Code Security | Developers First | SonarSource*, Jan. 2021. Adresse: <https://www.sonarsource.com/> (besucht am 06. 01. 2021).
 - [76] G. A. Campbell, *Cognitive Complexity - SonarSource*, SonarSource, Hrsg., Feb. 2017. Adresse: <https://www.sonarsource.com/docs/CognitiveComplexity.pdf> (besucht am 06. 01. 2021).
 - [77] J. Chase, S. Tkach und D. Timofeev, *jni*, 2020. Adresse: <https://crates.io/crates/jni> (besucht am 20. 01. 2021).

Anhang A: Einführung Rust

Da Rust eine relativ neue Sprache mit einer teilweise ungewöhnlichen Syntax im Vergleich zu C und dessen Ablegern ist, ist hier eine kurze Einführung zu finden. Damit soll es mit grundlegenden Programmierkenntnissen möglich sein, mindestens vorgestellten Code zu verstehen.

A.1 Grundlagen

Variablendeklaration Diese werden mit dem `let`-Schlüsselwort eingeleitet, gefolgt vom Variablennamen. Der Rust-Compiler versucht, die Datentypen von Variablen automatisch zu bestimmen. Daher kann normalerweise bei der Deklaration auf den Datentyp verzichtet werden. Die Variable kann beliebig oft neu angelegt werden. Soll eine Variable veränderbar sein, muss dies mit `mut` explizit angegeben werden.

```
———— Variablendeklaration ————
1 let mut x = 5; // erkannt: i32
2 let y: u8 = 20; // ueberschrieben: Typ ist u8 (unsigned
3                   // integer, 8 Bit)
4 x = 8; // nur Zuweisung (muss *mut* sein)
5 let y = "hallo"; // Neudeklaration
```

Zahlentypen In Rust haben die Zahlen-Typen einen Namen bestehend aus dem ersten Buchstaben des jeweiligen Typs (*i* für (signed) integer, *u* für unsigned und *f* für float), gefolgt von der Größe in Bits. Mögliche Typen sind `i8` bis `i128` (in Zweierpotenzen), `u8` bis `u128`, `f32` und `f64`. Außerdem existieren noch die beiden plattformspezifischen Typen `usize` und `isize`, deren Größe der Adressbreite des jeweiligen Systems entspricht. Standardmäßig werden `i32` bzw. `f64` angenommen.

Strings Es existieren zwei String-Typen: `String` und `&str` (*string slice*; Zeiger auf (Beginn eines) Abschnitt eines Strings). *Strings* sind echte Objekte mit Owner, `&str` hingegen sind Referenztypen.

```
———— Arten von Strings ————
1 let s1 = String::new(); // String
2 let s2 = "welt"; // &str
3 let s3 = s1.as_str(); // &str;
4 let s4 = String::from(s2); // String
```

Arrays Arrays haben wie in C eine feste Länge, die beim Kompilieren bekannt sein muss. Ein Array kann zu Beginn mit einem bestimmten Wert gefüllt werden.

```

Array-Deklarationen
1 let a: [i32; 5] = [1, 2, 3, 4, 5];
2 let b = [0; 5]; // [0, 0, 0, 0, 0]

```

Funktionen und Methoden Es wird unterschieden zwischen Funktionen, die statisch, also unabhängig von „Objekten“ (*structs*, *enums*, ...) sind, und Methoden, die für eines der genannten Objekte definiert werden und sich darauf beziehen. Die Syntax ist grundlegend gleich:

```

fn <Funktionsname>(<Parameter 1>: <Typ von Parameter 1>,
    <Parameter 2>: <Typ von Parameter 2>)
    [-> <Rückgabotyp>] {
}

```

Funktionen werden ggf. mit Modulname und `::` aufgerufen, Methoden stets mit einem `.` zwischen Objekt- und Methodenname.

```

Beispiel-Funktion
1 fn summe(a: i32, b: i32) -> i32 {
2     return a + b;
3 }
4
5 let x = summe(5, 3);

```

```

Beispiel-Methode mit struct
1 struct Example {
2     a: i32,
3     b: i32
4 }
5
6 impl Example {
7     fn summe(self) -> i32 {
8         return self.a + self.b;
9     }
10 }
11
12 let s = Example {
13     a: 5,
14     b: 3
15 };
16 let x = s.summe();

```

Anweisungen und Ausdrücke Alle Instruktionen in Rust sind entweder Anweisungen (*Statements*) oder Ausdrücke (*Expressions*). Anweisungen werden ausgeführt und geben dabei keinen Wert zurück. Beispiele hierfür sind Zuweisungen oder Funktionsdefinitionen. Ausdrücke hingegen haben einen bestimmten Wert, der beispielsweise einer Variablen zugewiesen werden kann. Konstanten sind somit Ausdrücke, genauso wie Funktionsaufrufe oder Blöcke. Der letzte Ausdruck einer Funktion oder eines Blocks stellt dabei den Wert des Ausdrucks und damit den Rückgabewert dar. [30, Kap. 3.3]

Beispiele:

```
Beispiele für Anweisungen und Ausdrücke
1 let x = 5; // Anweisung, kein Rückgabewert
2 //let x = (let y = 6); // nicht möglich, da Zuweisung eine
3 // Anweisung ist und damit nicht zugewiesen werden kann
4
5 let y = { // Blöcke sind Ausdrücke
6     let a = 1;
7     a + fuenf() // Ausdruck, Wert des Blocks
8 };
9
10 fn fuenf() -> i32 { // Funktionsdefinition: Anweisung
11     5 // Ausdruck; "return 5;"
12 }
```

Enums In einem *Enum* wird eine Menge möglicher Ausprägungen gespeichert. Jedes Element bekommt dabei einen Index zugewiesen, der auch explizit gesetzt werden kann. In Rust kann zusätzlich jede Möglichkeit mit eigenen Daten verknüpft werden.

```
Enums in Rust
1 enum Beispiel {
2     B1,
3     B2 = 5,
4 }
5
6 enum Beispiel2 {
7     B3(i32),
8 }
9
10 let x = Beispiel::B1; // x = B1
11 let y = Beispiel::B2 as i32; // y = 5
12 let z = Beispiel2::B3(20); // z = B3(20)
```

A.2 Besondere Strukturen

Makros Sie dienen der Vereinfachung von verschiedenen Anweisungen. Der Name endet stets mit einem Ausrufezeichen !.

Beispiele für Makros

```

1 println!("hallo, ");
2 println!("Welt.");
3 eprintln!("Hallo, stderr");
4 let x = format!("{} + {} = {}", 1, 2, 1 + 2);
5 panic!("Crash");
6 assert_eq!(5, 2 + 3);

```

match-Operator Dieser ersetzt in Rust die verbreitete *switch*-Anweisung und vergleicht eine Variable mit einer Reihe von Werten, den sog. Mustern. Eine Besonderheit ist dabei, dass sämtliche möglichen Werte geprüft werden müssen. Um das zu erreichen, existiert das besondere Muster `_`, welches alle übrigen Werte vereint. *match*-Blöcke sind *expressions* und können damit einer Variablen zugewiesen werden.

Beispiele mit match

```

1 enum JaNein {
2     Ja,
3     Nein,
4 }
5 let x = JaNein::Ja;
6 let x_string = match x {
7     JaNein::Ja => "ja",
8     JaNein::Nein => "nein",
9 };
10 assert_eq!(x_string, "ja");
11
12 let y: u8 = 5;
13 match y {
14     0 | 1 | 2 | 3 => println!("kleiner als vier"),
15     4 => {
16         println!("vier");
17         println!("Beispiel fuer mehrere Anweisungen");
18     },
19     _ => println!("groesser als vier"),
20 }

```

Enums Result und Option Diese beiden Enums sind typische Rückgabetypen von Funktionen. Eine Ausprägung ist in beiden Fällen der richtige Wert, die andere ist entweder ein Fehler (*Result*) oder „Nichts“ (*Option*). Der Erfolgswert kann dabei

unter anderem jeweils über die `unwrap()`-Methode extrahiert werden, wobei das Programm abstürzt, wenn `unwrap()` für die Fehlerwerte aufgerufen wird.

Eine andere Möglichkeit ist der `match`-Operator, eine *if*-Abfrage eine dritte.

Definition von `Result` und `Option`

```
1 enum Result<T, E> {  
2     Ok(T),  
3     Err(E),  
4 }  
5  
6 enum Option<T> {  
7     None,  
8     Some(T),  
9 }
```

Beispiele

```
1 /* unwrap */  
2 let x = Some(5); // Option<i32>  
3 let y = x.unwrap(); // y = 5  
4  
5 let x: Option<i32> = None;  
6 let y = x.unwrap(); // panic  
7  
8 /* match */  
9 let x = Some(5);  
10 match x {  
11     Some(num) => println!("x = {}", num),  
12     None => eprintln!("x has no value"),  
13 };  
14  
15 let file_result: Result<File, Error> =  
16     File::open("test.txt");  
17 match file_result {  
18     Ok(file) => println!("File opened successfully"),  
19     Err(e) => eprintln!("Could not open file: {}", e),  
20 };  
21  
22 /* if let */  
23 let x = Some(5);  
24 if let Some(num) = x {  
25     println!("x = {}", num);  
26 }
```

Anhang B: Tabellen

B.1 Kompilierzeiten Hello-World-Programme

Tabelle B.1: Kompilierzeiten eines *Hello-World*-Programmes in C mit *gcc*, gemessen mit *time*

Messung	real	user	sys
1	0m0,049s	0m0,038s	0m0,011s
2	0m0,046s	0m0,030s	0m0,016s
3	0m0,045s	0m0,030s	0m0,015s
4	0m0,047s	0m0,031s	0m0,016s
5	0m0,041s	0m0,030s	0m0,011s
6	0m0,047s	0m0,038s	0m0,009s
7	0m0,040s	0m0,025s	0m0,014s
8	0m0,046s	0m0,033s	0m0,013s
9	0m0,042s	0m0,019s	0m0,023s
10	0m0,046s	0m0,032s	0m0,014s
11	0m0,048s	0m0,022s	0m0,026s
12	0m0,049s	0m0,039s	0m0,010s
13	0m0,041s	0m0,034s	0m0,007s
14	0m0,044s	0m0,027s	0m0,017s
15	0m0,041s	0m0,025s	0m0,016s
16	0m0,046s	0m0,032s	0m0,014s
17	0m0,047s	0m0,035s	0m0,013s
18	0m0,046s	0m0,028s	0m0,018s
19	0m0,046s	0m0,032s	0m0,014s
20	0m0,045s	0m0,036s	0m0,010s
Durchschnitt	0,0451 s	0,0308 s	0,01435 s

Tabelle B.2: Kompilierzeiten eines *Hello-World*-Programmes in Java mit *javac*, gemessen mit *time*

Messung	real	user	sys
1	0m0,611s	0m1,496s	0m0,085s
2	0m0,501s	0m1,448s	0m0,122s
3	0m0,494s	0m1,446s	0m0,118s
4	0m0,506s	0m1,516s	0m0,058s
5	0m0,496s	0m1,436s	0m0,097s
6	0m0,464s	0m1,403s	0m0,073s
7	0m0,52s	0m1,542s	0m0,066s
8	0m0,469s	0m1,403s	0m0,056s

9	0m0,498s	0m1,414s	0m0,089s
10	0m0,477s	0m1,439s	0m0,097s
11	0m0,48s	0m1,376s	0m0,103s
12	0m0,489s	0m1,477s	0m0,076s
13	0m0,469s	0m1,341s	0m0,097s
14	0m0,481s	0m1,443s	0m0,077s
15	0m0,515s	0m1,511s	0m0,088s
16	0m0,506s	0m1,515s	0m0,049s
17	0m0,472s	0m1,403s	0m0,063s
18	0m0,506s	0m1,446s	0m0,07s
19	0m0,479s	0m1,421s	0m0,049s
20	0m0,518s	0m1,517s	0m0,076s
Durchschnitt	0,49755 s	1,44965 s	0,08045 s

Tabelle B.3: Kompilierzeiten eines *Hello-World*-Programmes in Rust mit *rustc*, gemessen mit *time*

Messung	real	user	sys
1	0m0,228s	0m0,198s	0m0,059s
2	0m0,229s	0m0,204s	0m0,056s
3	0m0,225s	0m0,196s	0m0,064s
4	0m0,224s	0m0,200s	0m0,057s
5	0m0,227s	0m0,203s	0m0,055s
6	0m0,225s	0m0,200s	0m0,057s
7	0m0,230s	0m0,189s	0m0,071s
8	0m0,228s	0m0,182s	0m0,076s
9	0m0,227s	0m0,200s	0m0,059s
10	0m0,225s	0m0,189s	0m0,071s
11	0m0,226s	0m0,211s	0m0,050s
12	0m0,222s	0m0,195s	0m0,059s
13	0m0,219s	0m0,158s	0m0,093s
14	0m0,226s	0m0,189s	0m0,068s
15	0m0,226s	0m0,193s	0m0,059s
16	0m0,234s	0m0,203s	0m0,063s
17	0m0,216s	0m0,189s	0m0,057s
18	0m0,222s	0m0,170s	0m0,080s
19	0m0,237s	0m0,198s	0m0,070s
20	0m0,223s	0m0,182s	0m0,074s
Durchschnitt	0,22595 s	0,19245 s	0,0649 s

B.2 Computer Language Benchmarks Game

Tabelle B.4: Ergebnisse für *binary-trees*-Benchmark, sortiert nach Zeit

Quelle: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/binarytrees.html>

Programm	Zeit (Sek.)	Speichernutzung (Seiten)	Größe (Bytes)
<i>Rust #5</i>	1.07	198540	765
<i>Rust #2</i>	1.21	200476	721
<i>Rust #4</i>	1.25	167028	756
<i>Rust #3</i>	1.35	169928	754
<i>C gcc #2</i>	1.79	168760	809
<i>C gcc #3</i>	2.02	133960	836
<i>Rust</i>	2.37	132892	751
<i>Java #7</i>	2.50	2487096	835
<i>Java #4</i>	4.20	2178184	840
<i>Java #6</i>	4.62	2002384	529
<i>Java #3</i>	4.65	2013384	540
<i>Java #2</i>	4.82	1952424	552
<i>C gcc #5</i>	8.40	351432	908
<i>C gcc</i>	21.00	263616	654

Tabelle B.5: Ergebnisse für *fannkuch-redux*-Benchmark, sortiert nach Zeit

Quelle: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/fannkuchredux.html>

Programm	Zeit (Sek.)	Speichernutzung (Seiten)	Größe (Bytes)
<i>Rust #5</i>	7.28	1168	1016
<i>C gcc #5</i>	7.52	836	910
<i>Rust #4</i>	7.69	1168	1020
<i>Rust #3</i>	7.69	1172	1020
<i>Rust #2</i>	9.94	980	1191
<i>C gcc #2</i>	10.85	912	1557
<i>Java</i>	11.00	34104	1282
<i>C gcc #4</i>	13.90	792	1183
<i>Java #3</i>	41.43	33864	1257
<i>C gcc</i>	41.69	740	508
<i>C gcc #3</i>	41.90	740	567
<i>Java #2</i>	44.81	34576	514

Tabelle B.6: Ergebnisse für *fasta*-Benchmark, sortiert nach Zeit

Quelle: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/fasta.html>

Programm	Zeit (Sek.)	Speichernutzung (Seiten)	Größe (Bytes)
<i>Rust #7</i>	0.77	1848	2529
<i>C gcc #2</i>	0.82	2896	2268
<i>C gcc #7</i>	0.84	2888	2011
<i>C gcc #6</i>	0.93	2376	1930
<i>Rust #5</i>	0.93	2328	1961
<i>Rust #2</i>	0.94	2384	1906
<i>Java #6</i>	1.20	44124	2543
<i>Java #5</i>	1.27	43968	2473
<i>C gcc #5</i>	1.91	804	1281
<i>Rust #3</i>	2.04	836	1233
<i>C gcc #4</i>	3.07	740	1240
<i>Java #4</i>	3.18	37560	1524
<i>C gcc</i>	3.70	792	1206
<i>Rust #6</i>	4.09	123788	955
<i>Rust #4</i>	4.15	123788	939
<i>Rust</i>	4.44	832	1237
<i>Java #2</i>	4.44	38128	1257

Tabelle B.7: Ergebnisse für *k-nucleotide*-Benchmark, sortiert nach Zeit

Quelle: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/knucleotide.html>

Programm	Zeit (Sek.)	Speichernutzung (Seiten)	Größe (Bytes)
<i>Rust #6</i>	2.76	159224	1691
<i>Rust #7</i>	3.14	131448	1578
<i>Rust #5</i>	3.81	133952	1739
<i>C gcc</i>	3.81	130112	1506
<i>Rust #8</i>	3.93	133404	1543
<i>Java</i>	5.00	408040	1812
<i>Rust #2</i>	6.30	137668	1347
<i>Rust #9</i>	6.59	134940	1841
<i>Rust #4</i>	6.86	136428	1749
<i>Java #6</i>	7.41	419660	1607
<i>Java #3</i>	7.43	419260	1635
<i>Java #5</i>	17.49	208408	2219
<i>Rust</i>	25.58	152008	1078
<i>Java #4</i>	36.02	198196	1882

Tabelle B.8: Ergebnisse für *mandelbrot*-Benchmark, sortiert nach Zeit

Quelle: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/mandelbrot.html>

Programm	Zeit (Sek.)	Speichernutzung (Seiten)	Größe (Bytes)
<i>Rust #8</i>	0.93	32788	763
<i>Rust #7</i>	0.93	32644	757
<i>Rust #3</i>	0.99	12920	1007
<i>Rust #5</i>	1.16	33608	719
<i>C gcc #6</i>	1.27	31692	1135
<i>Rust #6</i>	1.33	33768	1332
<i>Rust</i>	2.78	38744	868
<i>C gcc #8</i>	3.17	33104	782
<i>C gcc #4</i>	3.46	32832	799
<i>C gcc #7</i>	3.48	33048	994
<i>C gcc</i>	3.48	28740	822
<i>C gcc #3</i>	3.51	32440	763
<i>Java #2</i>	4.11	68204	796
<i>C gcc #9</i>	4.16	32800	694
<i>Java #6</i>	4.25	68636	802
<i>Java #4</i>	5.14	67456	660
<i>Java #3</i>	7.36	68692	903
<i>C gcc #2</i>	25.32	796	400
<i>Java</i>	27.74	34536	665

Tabelle B.9: Ergebnisse für *n-body*-Benchmark, sortiert nach Zeit

Quelle: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/nbody.html>

Programm	Zeit (Sek.)	Speichernutzung (Seiten)	Größe (Bytes)
<i>Rust #8</i>	3.31	836	1767
<i>Rust #7</i>	3.39	840	1753
<i>Rust #2</i>	3.75	840	1805
<i>Rust</i>	3.93	836	1480
<i>C gcc #8</i>	4.30	8	1391
<i>C gcc #4</i>	4.39	1060	1490
<i>Rust #5</i>	4.84	896	2220
<i>Rust #4</i>	4.91	832	1803
<i>Rust #6</i>	5.05	832	1790
<i>Rust #3</i>	5.25	840	1540
<i>C gcc #5</i>	6.09	796	1429
<i>C gcc #7</i>	6.35	1064	1244
<i>C gcc #6</i>	6.45	1120	1180
<i>C gcc #2</i>	6.62	1120	1263
<i>Java #4</i>	6.75	35200	1489

<i>C gcc</i>	6.88	1056	1173
<i>Java #5</i>	6.98	35560	1429
<i>C gcc #3</i>	7.18	1116	1208
<i>Java #2</i>	7.43	35516	1424
<i>Java #3</i>	7.45	35508	1430
<i>Java</i>	7.80	37588	1430

Tabelle B.10: Ergebnisse für *pidigits*-Benchmark, sortiert nach Zeit

Quelle: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/pidigits.html>

Programm	Zeit (Sek.)	Speichernutzung (Seiten)	Größe (Bytes)
<i>C gcc #2</i>	0.73	2848	416
<i>Rust</i>	0.74	2888	1420
<i>C gcc</i>	0.74	2564	452
<i>Rust #3</i>	0.74	2892	1366
<i>Rust #2</i>	0.74	2888	1306
<i>Java #3</i>	0.79	35568	764
<i>Java #2</i>	1.29	38736	938
<i>Java</i>	7.61	518172	800

Tabelle B.11: Ergebnisse für *regex-redux*-Benchmark, sortiert nach Zeit

Quelle: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/regexredux.html>

Programm	Zeit (Sek.)	Speichernutzung (Seiten)	Größe (Bytes)
<i>Rust #7</i>	0.78	146236	2458
<i>C gcc #5</i>	0.80	152196	1397
<i>C gcc #4</i>	0.94	152124	1230
<i>Rust #6</i>	1.30	153432	986
<i>Rust #5</i>	1.38	201892	771
<i>Rust #4</i>	1.40	199148	765
<i>Rust</i>	1.40	152736	804
<i>Rust #2</i>	1.42	149828	701
<i>Rust #3</i>	1.47	202472	719
<i>C gcc #3</i>	1.70	103532	1266
<i>C gcc #2</i>	3.18	102904	1115
<i>Java #3</i>	5.70	656328	929
<i>Java #6</i>	5.82	633348	740
<i>Java</i>	9.28	645032	868

Tabelle B.12: Ergebnisse für *reverse-complement*-Benchmark, sortiert nach Zeit

Quelle: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/revcomplement.html>

Programm	Zeit (Sek.)	Speichernutzung (Seiten)	Größe (Bytes)
<i>Rust</i>	0.46	499024	3040
<i>Rust #2</i>	0.76	995032	1330
<i>Rust #3</i>	0.79	995112	1097
<i>C gcc #6</i>	0.86	698264	820
<i>C gcc #2</i>	0.98	994220	750
<i>C gcc #5</i>	1.41	994048	647
<i>Java #8</i>	1.54	702332	2183
<i>Java #3</i>	2.44	1172748	1722
<i>Java #6</i>	2.94	2054460	752
<i>C gcc #4</i>	3.40	500236	709
<i>Java #5</i>	4.44	1122300	1108
<i>Java #4</i>	4.87	1925516	651
<i>Java #7</i>	22.74	1032516	1647

Tabelle B.13: Ergebnisse für *spectralnorm*-Benchmark, sortiert nach Zeit

Quelle: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/spectralnorm.html>

Programm	Zeit (Sek.)	Speichernutzung (Seiten)	Größe (Bytes)
<i>Rust #5</i>	0.71	2696	1055
<i>Rust #4</i>	0.72	2596	817
<i>C gcc #5</i>	0.72	1100	569
<i>Rust #6</i>	0.72	2524	1126
<i>C gcc #4</i>	0.72	2356	1139
<i>Rust #2</i>	0.73	2276	1110
<i>Rust</i>	0.73	2224	1254
<i>Rust #7</i>	0.73	2576	932
<i>Rust #3</i>	0.75	2228	1053
<i>C gcc #3</i>	1.43	1100	463
<i>Java #2</i>	1.68	38348	950
<i>Java #3</i>	1.68	38984	756
<i>C gcc</i>	5.39	1052	383
<i>Java</i>	6.17	38168	514

B.3 Kompilierzeiten Webserver

B.3.1 C

Tabelle B.14: C-Webserver, kompiliert mit *gcc*, Version: Debug ohne Optimierungen; Messung mit *time*

Ausgeführter Befehl:

```
time /usr/bin/gcc -DDEBUG -g -Wall -Wextra -Werror -pthread -o
bin/webserver_opt_gcc src/*.c -L/usr/lib -lssl -lcrypto
```

Messung	real	user	sys
1	0m0,619s	0m0,475s	0m0,141s
2	0m0,612s	0m0,459s	0m0,153s
3	0m0,604s	0m0,495s	0m0,108s
4	0m0,616s	0m0,457s	0m0,158s
5	0m0,614s	0m0,448s	0m0,165s
6	0m0,591s	0m0,462s	0m0,128s
7	0m0,592s	0m0,462s	0m0,128s
8	0m0,617s	0m0,502s	0m0,114s
9	0m0,603s	0m0,468s	0m0,133s
10	0m0,628s	0m0,507s	0m0,120s
11	0m0,582s	0m0,455s	0m0,126s
12	0m0,620s	0m0,453s	0m0,167s
13	0m0,612s	0m0,475s	0m0,137s
14	0m0,623s	0m0,475s	0m0,147s
15	0m0,613s	0m0,477s	0m0,135s
16	0m0,611s	0m0,483s	0m0,128s
17	0m0,610s	0m0,468s	0m0,140s
18	0m0,607s	0m0,453s	0m0,154s
19	0m0,594s	0m0,462s	0m0,131s
20	0m0,612s	0m0,473s	0m0,138s
Durchschnitt	0 m 0,609 s	0 m 0,47 s	0 m 0,138 s

Tabelle B.15: C-Webserver, kompiliert mit *gcc*, Version: Debug mit Optimierungen (-Og); Messung mit *time*

Ausgeführter Befehl:

```
time /usr/bin/gcc -DDEBUG -g -Og -Wall -Wextra -Werror -pthread
-o bin/webserver_opt_gcc src/*.c -L/usr/lib -lssl -lcrypto
```

Messung	real	user	sys
1	0m0,752s	0m0,632s	0m0,119s
2	0m0,777s	0m0,598s	0m0,178s
3	0m0,765s	0m0,654s	0m0,110s

4	0m0,796s	0m0,666s	0m0,129s
5	0m0,793s	0m0,650s	0m0,142s
6	0m0,795s	0m0,644s	0m0,149s
7	0m0,785s	0m0,687s	0m0,098s
8	0m0,778s	0m0,618s	0m0,159s
9	0m0,795s	0m0,654s	0m0,140s
10	0m0,783s	0m0,645s	0m0,137s
11	0m0,788s	0m0,629s	0m0,158s
12	0m0,762s	0m0,656s	0m0,104s
13	0m0,776s	0m0,645s	0m0,131s
14	0m0,784s	0m0,639s	0m0,144s
15	0m0,779s	0m0,669s	0m0,109s
16	0m0,780s	0m0,600s	0m0,178s
17	0m0,793s	0m0,632s	0m0,160s
18	0m0,768s	0m0,621s	0m0,146s
19	0m0,787s	0m0,655s	0m0,131s
20	0m0,791s	0m0,645s	0m0,144s
Durchschnitt	0 m 0,781 s	0 m 0,642 s	0 m 0,138 s

Tabelle B.16: C-Webserver, kompiliert mit *gcc*, Version: Release mit Optimierungen; Messung mit *time*

Ausgeführter Befehl:

```
time /usr/bin/gcc -O3 -ffast-math -flto -Wall -Wextra -Werror
-pthread -o bin/webserver_opt_gcc src/*.c -L/usr/lib -lssl
-lcrypto
```

Messung	real	user	sys
1	0m0,887s	0m0,765s	0m0,121s
2	0m0,912s	0m0,750s	0m0,161s
3	0m0,872s	0m0,708s	0m0,163s
4	0m0,880s	0m0,709s	0m0,170s
5	0m0,879s	0m0,731s	0m0,147s
6	0m0,876s	0m0,726s	0m0,148s
7	0m0,904s	0m0,763s	0m0,141s
8	0m0,865s	0m0,731s	0m0,131s
9	0m0,885s	0m0,723s	0m0,162s
10	0m0,889s	0m0,731s	0m0,158s
11	0m0,881s	0m0,704s	0m0,176s
12	0m0,900s	0m0,711s	0m0,189s
13	0m0,914s	0m0,717s	0m0,196s
14	0m0,891s	0m0,765s	0m0,126s
15	0m0,890s	0m0,741s	0m0,148s
16	0m0,902s	0m0,747s	0m0,154s

17	0m0,894s	0m0,752s	0m0,141s
18	0m0,884s	0m0,724s	0m0,160s
19	0m0,909s	0m0,753s	0m0,155s
20	0m0,908s	0m0,745s	0m0,162s
Durchschnitt	0 m 0,891 s	0 m 0,735 s	0 m 0,155 s

Tabelle B.17: C-Webserver, kompiliert mit *clang*, Version: Debug ohne Optimierungen; Messung mit *time*

Ausgeführter Befehl:

```
time /usr/bin/clang -DDEBUG -g -Wall -Wextra -Werror -pthread
-o bin/webserver_opt_clang src/*.c -L/usr/lib -lssl -lcrypto
```

Messung	real	user	sys
1	0m0,719s	0m0,585s	0m0,111s
2	0m0,695s	0m0,572s	0m0,123s
3	0m0,692s	0m0,552s	0m0,140s
4	0m0,703s	0m0,564s	0m0,138s
5	0m0,700s	0m0,572s	0m0,127s
6	0m0,696s	0m0,545s	0m0,151s
7	0m0,694s	0m0,564s	0m0,130s
8	0m0,697s	0m0,544s	0m0,153s
9	0m0,709s	0m0,554s	0m0,153s
10	0m0,714s	0m0,598s	0m0,115s
11	0m0,716s	0m0,544s	0m0,171s
12	0m0,701s	0m0,535s	0m0,165s
13	0m0,687s	0m0,524s	0m0,163s
14	0m0,694s	0m0,546s	0m0,148s
15	0m0,707s	0m0,538s	0m0,168s
16	0m0,703s	0m0,552s	0m0,151s
17	0m0,707s	0m0,507s	0m0,199s
18	0m0,681s	0m0,531s	0m0,149s
19	0m0,700s	0m0,561s	0m0,138s
20	0m0,696s	0m0,554s	0m0,142s
Durchschnitt	0 m 0,701 s	0 m 0,552 s	0 m 0,147 s

Tabelle B.18: C-Webserver, kompiliert mit *clang*, Version: Release mit Optimierungen; Messung mit *time*

Ausgeführter Befehl:

```
time /usr/bin/clang -O3 -ffast-math -flto -Wall -Wextra -Werror
-pthread -o bin/webserver_opt_clang src/*.c -L/usr/lib -lssl
-lcrypto
```

Messung	real	user	sys
1	0m0,951s	0m0,744s	0m0,187s
2	0m0,920s	0m0,757s	0m0,161s
3	0m0,928s	0m0,794s	0m0,133s
4	0m0,922s	0m0,761s	0m0,161s
5	0m0,919s	0m0,755s	0m0,163s
6	0m0,937s	0m0,785s	0m0,151s
7	0m0,907s	0m0,784s	0m0,122s
8	0m0,927s	0m0,717s	0m0,210s
9	0m0,931s	0m0,783s	0m0,147s
10	0m0,932s	0m0,781s	0m0,150s
11	0m0,924s	0m0,756s	0m0,167s
12	0m0,923s	0m0,796s	0m0,126s
13	0m0,926s	0m0,794s	0m0,132s
14	0m0,924s	0m0,755s	0m0,168s
15	0m0,938s	0m0,792s	0m0,146s
16	0m0,922s	0m0,769s	0m0,152s
17	0m0,942s	0m0,797s	0m0,143s
18	0m0,897s	0m0,761s	0m0,135s
19	0m0,934s	0m0,780s	0m0,153s
20	0m0,923s	0m0,766s	0m0,156s
Durchschnitt	0 m 0,926 s	0 m 0,771 s	0 m 0,153 s

B.3.2 Java

Tabelle B.19: Java-Webserver, kompiliert mit *javac/gradle*; Messung mit *time*

Ausgeführter Befehl:

```
./gradlew clean; time ./gradlew assemble
```

Messung	real	user	sys
1	0m1,347s	0m1,519s	0m0,170s
2	0m1,266s	0m1,524s	0m0,114s
3	0m1,229s	0m1,526s	0m0,129s
4	0m1,289s	0m1,481s	0m0,082s
5	0m1,246s	0m1,488s	0m0,128s
6	0m1,250s	0m1,606s	0m0,099s

7	0m1,199s	0m1,528s	0m0,113s
8	0m1,202s	0m1,590s	0m0,113s
9	0m1,163s	0m1,494s	0m0,127s
10	0m1,215s	0m1,523s	0m0,125s
11	0m1,183s	0m1,453s	0m0,096s
12	0m1,188s	0m1,601s	0m0,129s
13	0m1,187s	0m1,536s	0m0,114s
14	0m1,145s	0m1,424s	0m0,111s
15	0m1,176s	0m1,567s	0m0,111s
16	0m1,151s	0m1,528s	0m0,132s
17	0m1,193s	0m1,433s	0m0,091s
18	0m1,188s	0m1,524s	0m0,107s
19	0m1,173s	0m1,497s	0m0,077s
20	0m1,156s	0m1,482s	0m0,139s
Durchschnitt	0 m 1,207 s	0 m 1,516 s	0 m 0,115 s

B.3.3 Rust

Tabelle B.20: Rust-Webserver, kompiliert mit *rustc/cargo*, Version: Debug ohne Optimierungen;
Messung mit *time*

Ausgeführter Befehl:

`time cargo build`

Messung	real	user	sys
1	0m2,078s	0m2,910s	0m0,643s
2	0m2,058s	0m2,808s	0m0,678s
3	0m2,073s	0m2,946s	0m0,587s
4	0m2,058s	0m2,835s	0m0,655s
5	0m2,075s	0m2,817s	0m0,686s
6	0m2,056s	0m2,971s	0m0,536s
7	0m2,134s	0m2,947s	0m0,624s
8	0m2,105s	0m2,869s	0m0,646s
9	0m2,097s	0m2,870s	0m0,677s
10	0m2,051s	0m2,840s	0m0,656s
11	0m2,051s	0m2,867s	0m0,600s
12	0m2,041s	0m2,848s	0m0,601s
13	0m2,064s	0m2,965s	0m0,545s
14	0m2,071s	0m2,909s	0m0,625s
15	0m2,079s	0m2,852s	0m0,654s
16	0m2,080s	0m2,880s	0m0,644s
17	0m2,073s	0m2,831s	0m0,653s
18	0m2,047s	0m2,873s	0m0,623s

19	0m2,099s	0m2,839s	0m0,680s
20	0m2,066s	0m2,833s	0m0,626s
Durchschnitt	0 m 2,073 s	0 m 2,876 s	0 m 0,632 s

Tabelle B.21: Rust-Webserver, kompiliert mit *rustc/cargo*, Version: Release mit Optimierungen;
Messung mit *time*

Ausgeführter Befehl:

`time cargo build --release`

Messung	real	user	sys
1	0m3,246s	0m9,144s	0m0,286s
2	0m3,238s	0m9,100s	0m0,300s
3	0m3,258s	0m9,266s	0m0,272s
4	0m3,248s	0m9,351s	0m0,258s
5	0m3,274s	0m9,171s	0m0,270s
6	0m3,260s	0m9,259s	0m0,271s
7	0m3,245s	0m9,105s	0m0,298s
8	0m3,411s	0m9,296s	0m0,281s
9	0m3,278s	0m9,118s	0m0,264s
10	0m3,268s	0m9,190s	0m0,294s
11	0m3,244s	0m9,067s	0m0,275s
12	0m3,267s	0m9,208s	0m0,297s
13	0m3,257s	0m9,232s	0m0,211s
14	0m3,253s	0m9,050s	0m0,311s
15	0m3,236s	0m9,143s	0m0,255s
16	0m3,259s	0m9,093s	0m0,290s
17	0m3,288s	0m9,198s	0m0,279s
18	0m3,276s	0m9,226s	0m0,267s
19	0m3,253s	0m9,153s	0m0,282s
20	0m3,277s	0m9,130s	0m0,292s
Durchschnitt	0 m 3,267 s	0 m 9,175 s	0 m 0,278 s

Tabelle B.22: Rust-Webserver, kompiliert mit *rustc/cargo*, Version: Debug ohne Optimierungen,
inkl. Abhängigkeiten; Messung mit *time*

Ausgeführter Befehl:

`cargo clean; time cargo build`

Messung	real	user	sys
1	0m15,116s	1m44,986s	0m8,441s
2	0m15,051s	1m44,003s	0m8,411s
3	0m15,124s	1m44,801s	0m8,120s
4	0m15,373s	1m44,922s	0m8,189s

5	0m15,353s	1m44,144s	0m8,269s
6	0m15,239s	1m45,318s	0m8,162s
7	0m15,228s	1m44,128s	0m8,749s
8	0m15,525s	1m44,190s	0m8,579s
9	0m15,253s	1m44,744s	0m8,269s
10	0m15,438s	1m45,297s	0m8,124s
11	0m15,401s	1m44,814s	0m8,070s
12	0m15,233s	1m45,149s	0m8,085s
13	0m15,387s	1m44,607s	0m8,271s
14	0m15,399s	1m45,508s	0m8,301s
15	0m15,580s	1m45,695s	0m8,629s
16	0m15,560s	1m45,433s	0m8,510s
17	0m15,527s	1m44,908s	0m8,660s
18	0m15,704s	1m44,598s	0m8,413s
19	0m15,109s	1m45,338s	0m8,029s
20	0m15,180s	1m44,806s	0m8,433s
Durchschnitt	0 m 15,339 s	1 m 44,869 s	0 m 8,336 s

Tabelle B.23: Rust-Webserver, kompiliert mit *rustc/cargo*, Version: Release mit Optimierungen, inkl. Abhängigkeiten; Messung mit *time*

Ausgeführter Befehl:

`cargo clean; time cargo build --release`

Messung	real	user	sys
1	0m13,030s	1m1,899s	0m4,214s
2	0m15,364s	1m44,551s	0m8,359s
3	0m15,064s	1m44,049s	0m8,304s
4	0m15,015s	1m44,504s	0m8,532s
5	0m15,285s	1m44,359s	0m8,475s
6	0m15,421s	1m45,954s	0m8,230s
7	0m15,522s	1m45,361s	0m8,367s
8	0m15,428s	1m44,118s	0m8,395s
9	0m15,241s	1m44,898s	0m7,866s
10	0m15,662s	1m46,102s	0m8,327s
11	0m15,357s	1m45,652s	0m8,303s
12	0m15,665s	1m45,469s	0m8,354s
13	0m15,434s	1m45,752s	0m8,322s
14	0m15,428s	1m45,046s	0m8,476s
15	0m15,505s	1m45,824s	0m8,134s
16	0m15,615s	1m45,853s	0m8,371s
17	0m15,849s	1m46,176s	0m8,378s
18	0m15,327s	1m45,179s	0m8,517s

19	0m15,342s	1m45,297s	0m8,300s
20	0m15,220s	1m45,395s	0m8,379s
Durchschnitt	0 m 15,289 s	1 m 43,072 s	0 m 8,13 s

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 26. Januar 2021