Technische Hochschule Ingolstadt

Nutzen der Programmiersprache Rust

Thomas Buchard

Professionelle Textsatzsysteme

Erstprüfer Dr.-Ing. Paul Spannaus
Zweitprüfer M. Sc. Christoph Nebl
Ausgabedatum 18.03.2021
Abgabedatum 04.06.2021

Abstract *- Diese Arbeit beschäftigt sich mit der Programmiersprache Rust. Es werden einige Kernkonzepte vorgestellt und deren Vorteile herausgearbeitet. Der aktuelle Stand genutzter Programmiersprachen wird beleuchtet und ein Vergleich zu anderen Sprachen mit ähnlicher Abstraktionsebene und Effizienz kann gezogen werden. Hier wird, wegen der großen Ähnlichkeit, maßgeblich C/C++ aber auch Go als Vergleich dienen. Auch sehen wir uns an welche Möglichkeiten sich aus der besonderen Beschaffenheit von Rust ergeben und welche Szenarien dadurch sinnvoll durch Rust abgedeckt werden können.*
Mit Blick auf die Zukunft mutmaßen wir, in welche Bereiche Rust noch vordringen wird.

1. Einleitung

Seit Einführung der Programmiersprache C Anfang der Siebziger Jahre und später C++ gelten diese als de facto Standard bei systemnaher Programmierung. Warum das so ist und welche Vor- und Nachteile sich daraus ableiten, werden wir in dieser Arbeit kurz beleuchten, um beurteilen zu können, wie Rust im Vergleich einzuordnen ist und in wie weit es als Ablöse für C/C++ dienen könnte.
Als sie bei der Stackoverflow Programmierer Umfrage 2020 nach der populärsten Sprache gefragt wurden, antworteten 21,8% der Teilnehmer mit C und nur 5,1% mit Rust. Gefragt nach der meist geliebten Technologie jedoch, war die häufigste Antwort nun schon das fünfte Jahr in Folge Rust. [1] Es scheint also einen starken Bedarf nach einer systemnahen Technologie zu geben, der von den aktuell eingesetzten Sprachen nicht gedeckt wird. Worin sich dieser Bedarf äußert und welche Besonderheiten Rust auszeichnen, wird im Folgenden erläutert.

1.1. Motivation

Es hat sich in den letzten Jahrzehnten kaum etwas daran geändert, welche Technologien für die systemnahe Programmierung eingesetzt werden. Beim TIOBE Index, der ein Indikator für die Popularität von Programmiersprachen ist, konnte C im Mai 2021 abermals den ersten Platz erreichen, C++ Platz 4. [2]
Überall dort, wo Schnittstellen zur Hardware notwendig sind, braucht es Eigenschaften, die nur von wenigen Sprachen erfüllt werden können.

Seit seiner Einführung vor fast 50 Jahren hat es keine Sprache geschafft C nachhaltig zu verdrängen. Ob, und falls ja, in welchen Bereichen Rust eine Chance hat Marktanteile von C/C++ zu gewinnen, soll hier untersucht werden.

1.2. Forschungsfrage

Welche Alleinstellungsmerkmale hat Rust im Vergleich zu anderen Programmiersprachen und wie sind diese zu bewerten? Welche Anwendungszwecke und Vorteile ergeben sich daraus?

1.3. Vorgehen und Untersuchungsmethoden

Es wird der aktuelle Stand der Technik überprüft. Anhand von Beispielen wird dargestellt welche Auswirkungen die Besonderheiten von Rust in der Praxis haben.

2. Stand der Technik

Es wird untersucht, welche Eigenschaften Rust besitzt und welche Einsatzgebiete sich daraus ableiten. Andere Technologien, die einen ähnlichen Zweck verfolgen, werden ebenfalls betrachtet, um einen Vergleich zu ermöglichen.

2.1. Überblick aktuell genutzter Programmiersprachen

Eine gute Einschätzung, welche Technologien zurzeit beliebt sind, bietet der TIOBE Index und die Stackoverflow Developer Survey. Laut diesen Quellen sind C/C++, Python, Java, C#, HTML/CSS, JavaScript und PHP stets unter den ersten Plätzen der letzten Jahre vertreten. Wie sich diese Sprachen untereinander unterscheiden und welche Vorteile sie bieten sehen wir im nächsten Punkt.

2.2. Klassifizierungen

Programmiersprachen lassen sich nach unterschiedlichen Kriterien gruppieren, beispielsweise ob sie statisch(z.B. Java) oder dynamisch(z.B. Python) typisiert sind, also ob Variablen beim Schreiben vom Code schon einem Datentyp zugeordnet werden oder ob die Laufzeitumgebung einen Typ aus dem Kontext heraus annimmt.
Eine weitere Abgrenzung ist die Objektorientierung, ein Konstrukt, das Datentypen erlaubt ihre eigenen Werte zu verändern, Methoden auf sich selbst anzuwenden und weitere Objekte zu generieren.
Zur Einordnung des Einsatzzwecks jedoch ist vor Allem die Art der Ausführung des Codes ein zentrales Unterscheidungsmerkmal. Die drei wichtigsten sind wie folgt:

- Interpretierte Sprachen werden von einem Interpreter direkt ausgeführt (Python, JavaScript). Der Interpreter bietet eine Laufzeitumgebung und kümmert sich um Speichermanagement.

- Bei kompilierten Sprachen mit Garbage Collector(Java, C#) muss geschriebener Code erst übersetzt werden, bevor er ausführbar wird. Ein Compiler setzt aus dem Code ein Programm zusammen, das von der Laufzeitumgebung (Java: Java Runtime Environment (JRE), C#: CLR)) ausgeführt werden kann. Auch hier werden Speicherzugriffe durch die Laufzeitumgebung koordiniert. Diese Koordinierung kostet aber nicht nur Geschwindigkeit, sondern ist auch nicht-deterministisch.

- Wenn also Performance und Echtzeitfähigkeit im Vordergrund stehen, muss auf eine Laufzeitumgebung verzichtet werden und der Speicher manuell verwaltet werden. Das funktioniert durch direktes Kompilieren des Codes in Maschinencode. Bei Sprachen wie C/C++ oder Rust setzt ein Compiler alle Teile des Codes zu einem Programm zusammen, das auf jedem PC der gleichen Prozessorarchitektur direkt ausgeführt werden kann. Das führt uns zu dem nächsten Punkt, dem Speichermanagement.

2.3. Speichermanagement

Betrachtet man die vorherigen Punkte der Klassifizierung von Sprachen merkt man schnell, dass für systemnahe Programmierung nur kompilierte Sprachen ohne Laufzeitumgebung in Frage kommen.
Nicht-deterministisches Verhalten und Overhead durch Garbage Collection haben in einer Umgebung, in der Geschwindigkeit eine große Rolle spielt keinen Platz. Prozess Scheduling wäre nicht effizient zu gestalten, da nicht genau vorhersagbar oder steuerbar ist, wann Garbage Collectoren arbeiten.
Wenn direkte Schnittstellen zur Hardware programmiert werden müssen, also Betriebssysteme, Treiber oder eingebettete Systeme, dann ist die direkte Übersetzung in Maschinencode meist unumgänglich. Das daraus nötige manuelle Handhaben der Speicherzugriffe bringt zwar viele Vorteile mit sich, birgt aber auch einige Herausforderungen.
Zum einen die Entwicklungsgeschwindigkeit. Manuelles Speicher Handling ist aufwendig. Sprachen mit automatischer Speicherbereinigung nehmen es einem beispielsweise ab, sich Gedanken machen zu müssen, zu welchem Zeitpunkt welche Variable freigegeben werden muss, damit sie nur genau ein mal freigegeben wird.
Es gibt zwar Konzepte, die einen dabei unterstützen, beispielsweise Ressourcenbelegung ist Initialisierung(RAII), das hauptsächlich bei C++ zum Einsatz kommt. Hier wird durch den Programmablauf darauf geachtet, dass Ressourcen erst erfolgreich initialisiert werden können, wenn sie belegt sind. Dies ermöglicht das Freigeben, sobald der Anwendungsbereich verlassen wird. Trotzdem ist der Entwicklungsaufwand deutlich höher als mit Garbage Collected Sprachen.
Dazu kommt die Fehleranfälligkeit. Auch unter Berücksichtigung aller gängigen Methoden zur Vermeidung von Fehlern treten selbst in großen und professionellen Projekten zahlreiche Speicherfehler auf.
Das Chromium Project veröffentlichte 2020 eine Statistik [3], die besagt, dass 70% der schwerwiegenden Sicherheitslücken auf unsichere Speicherzugriffe, also Fehler bei C/C++ Pointern, zurückzuführen sind. Microsoft liefert hier ähnliche Werte [4]. Genau diese Fehler versucht Rust gänzlich zu vermeiden.

2.4. Offizielle Ziele von Rust

Rust, das zum ersten Mal 2010 aufgetaucht ist, listet auf der offiziellen Rust Foundation Webseite [7] bei den häufig gestellten Fragen, Ziel des Projekts sei eine sichere, nebenläufige und praktikable Systemsprache zu implementieren. Rust existiere, weil ähnlich abstrakte und effiziente Sprachen nicht die folgenden Erwartungen erfüllen:

- Es wird zu wenig auf Sicherheit geachtet.
- Die Unterstützung für Nebenläufigkeit ist mangelhaft.
- Für manche Sprachmerkmale gibt es keine praktische, einleuchtende Benutzung.
- Es gibt nur eingeschränkte Kontrolle über Ressourcen.

Wie Rust diese Ziele erreichen will und welche Besonderheiten es sonst noch hat sehen wir im nächsten Punkt.

2.5. Alleinstellungsmerkmale

Ein Feature, das Laufzeitfehler vermeiden soll und stattdessen beim Kompilieren schon auf Fehler aufmerksam macht ist, dass Rust standardmäßig das Handling vom Null Fall erwartet.

In Listing 2.1 sehen wir eine Funktion, die eine Option übergeben bekommt. Diese Option enthält entweder einen String oder nichts. Die match Funktion geht auf beide Fälle ein, der Null Fall ist also abgedeckt.

Zudem kommt Rust mit einem Ökosystem, das die Entwicklung vereinfachen soll. [5] Dazu gehört

- rustup, ein Tool zur Installation von einer oder mehreren Rust Instanzen
- cargo, Tool das mit rust kommt und unter Anderem Abhängigkeiten auflöst, Tests ausführt und Dokumentation generiert.
- crates.io, die Community Seite zum Austausch von Rust Bibliotheken
- docs.rs, wo Dokumentationen von allen Bibliotheken von crates.io veröffentlicht werden

Listing 2.1: Rust Option, Some oder None [5]

```
fn greet_user(name: Option<String>) {
match name {
        Some(name) => println!("Hello there, {}!", name),
        None => println!("Well howdy, stranger!"),
}
}
```

- rustfmt, ein Formatter, der für Code nach Vorgaben des Rust style guides [6] sorgt
- clippy, ein linting Tool, also ein Werkzeug zur statischen Code Analyse
- rls, der Rust Language Server, der rustfmt, clippy und rust-analyzer zusammenbringt und in der IDE verfügbar macht

Rust´s beeindruckendstes Feature jedoch ist das Konzept von Ownership, zu deutsch Eigentum. Dadurch ist es möglich, Sicherheit beim Speicherzugriff zu garantieren, ohne einen Garbage Collector, der zusätzliche Ressourcen bei der Laufzeit benötigen würde. Das offizielle Rust Buch [7] erklärt das Prinzip anhand von drei Regeln:

- Jeder Wert in Rust hat eine Variable Eigentümer
- Es kann zu jeder Zeit nur einen Eigentümer geben
- Wenn der Eigentümer des Werts den Anwendungsbereich verlässt, wird der Wert freigegeben

Listing 2.2: Unzulässige Operation in Rust [11]

```
let s1 = String::from("hello");
let s2 = s1;
println!("{}, world!", s1);
```

Daraus ergeben sich Situationen, die ungewohnt erscheinen, wenn man noch keinen Kontakt mit Rust hatte. Listing 2.2 zeigt einen Code-Ausschnitt, bei dem ein String erzeugt wird und in s1 gespeichert. Dieser wird in s2 übertragen. Versucht man nun s1 zu benutzen, wird der Compiler eine Fehlermeldung ausgeben und das weitere Kompilieren verhindern.

Abbildung 2.1 zeigt, was passiert. Rust kopiert im Normalfall nur den Pointer im Stack der auf eine Adresse im Heap zeigt. Würden zwei Pointer, die

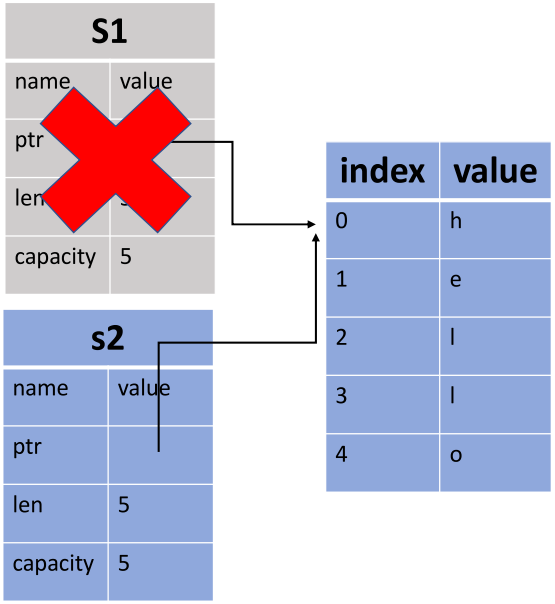

Abbildung 2.1.: Rust Shallow Copy [11]

auf den selben Wert zeigen, gleichzeitig freigegeben, würde das zu Fehlern in der Laufzeit führen. Um dieses Problem zu lösen, wird s1 für ungültig erklärt und löst somit dieses Problem. Zu jedem Zeitpunkt kann ein Wert nur einen Eigentümer haben. Ansonsten könnte man nicht mehr vorhersehen, wann der Speicher freigegeben werden muss.

Weil man meist nicht Eigentümerschaft übernehmen will beim Zugriff auf Daten, gibt es das Konzept des Borrowing, auf deutsch Leihen. Dazu gibt es einige Regeln, beispielsweise, dass nur eine mutable (veränderbare) geliehene Instanz von einem Wert gleichzeitig existieren darf. Auch darf es keine hängenden Zeiger geben. Der Borrow Checker stellt sicher, dass der Wert, auf den gezeigt wird, solange gültig bleibt bis auch der Zeiger aus dem Anwendungsbereich geht.
Diese Maßnahmen stellen sicher, dass Speicherzugriffe und Speicherfreigaben stets fehlerfrei ablau-

fen und Wettlaufbedingungen verhindert werden.

3. Beurteilung

In diesem Kapitel werden wir betrachten, in welchem Kontext das Programmieren in Rust vorteilhaft sein kann. Pro und Contra werden abgewägt und der Vergleich zu anderen Sprachen gezogen.

3.1. Vergleich zu anderen Sprachen

Wir vergleichen Rust mit einer kleinen Auswahl an anderen Programmiersprachen. Der Vergleich mit C/C++ ist besonders deshalb interessant, weil es viele Ähnlichkeiten zu Rust hat und deshalb auch mit Rust in Konkurrenz steht.
Doch auch ein Blick auf Go lohnt sich. Immerhin wurde auch Go 2009 ins Leben gerufen, um eine Alternative zu C++ zu schaffen [8]. Seither erfreut es sich wachsender Beliebtheit und bedient Bereiche, die sich mit Rust überlappen. Weil beide Sprachen auch vergleichsweise neu sind im Vergleich zu C/C++, werden sie auch oft im gleichen Atemzug genannt und verglichen.

Vergleich zu C/C++: Rust und C/C++ haben viele Gemeinsamkeiten. Beide werden kompiliert, haben keinen Garbage Collector, haben ähnlich schnelle Ausführgeschwindigkeit und erlauben Zugriff auf hardwarenahe Ressourcen. C++ ist wie auch Rust objektorientiert. Ihr Einsatzzweck überschneidet sich daher zu einem großen Teil, beide werden zur Programmierung von Betriebssystemen, Eingebetteten Systemen, Treibern und Anwendungen mit Fokus auf Geschwindigkeit und Ressourceneffizienz genutzt. Dabei profitiert C/C++ enorm von seiner großen Standardbibliothek und Zahl der Nutzer. Dadurch, dass es schon so lange existiert und es sich stets größter Beliebtheit erfreute, konnte sich ein riesiges Ökosystem entwickeln, das einem den Einstieg erleichtert und für jeden Anwendungszweck bereits ein Framework bereitstellt.
Wer also auf bestehende Schnittstellen und die große Community angewiesen ist, für den führt momentan noch kein Weg an C/C++ vorbei. Abgesehen davon bietet Rust durch das Konzept von Ownership und dem Borrow Checker große Vorteile bei Speichersicherheit und Fehleranfälligkeit.

Vergleich zu Go: Auch Go ist eine relativ junge Sprache, die 2009 von Google veröffentlicht wurde. Ihre Ziele unterscheiden sich jedoch ein wenig zu denen von Rust. Beide werden kompiliert und sind statisch typisiert, auch beim Fokus auf Nebenläufigkeit und garantierte Speichersicherheit überschneiden sich beide Sprachen. Effizienz in der Entwicklung spielt jedoch bei Go eine ebenso wichtige Rolle wie die Effizienz bei der Ausführung.
Der Compiler ist deutlich schneller als bei anderen Sprachen [8], um die Entwicklungszeit zu verkürzen. Aus dem selben Grund wird auch auf einen Garbage Collector gesetzt, der einem die Arbeit beim Speichermanagement abnimmt.
Dadurch ist Go für Projekte, die Zugriff auf hardwarenahe Komponenten benötigen nicht geeignet und konkurriert nicht mit Rust. Bei Software wie Webservern hingegen, bei der schnelle Veränderungen notwendig werden können und Ressourceneffizienz eine niedrigere Priorität als schnelle Anpassbarkeit hat, ist Go eine starke Konkurrenz zu Rust.

3.2. Szenarien in denen Rust Sinn ergibt

Eine systemnahe Programmiersprache wie Rust braucht man hauptsächlich dort, wo Performance wichtig ist oder Zugriff auf Hardwareressourcen notwendig. Dieser Bereich ist bisher fast vollständig von C/C++ abgedeckt worden. Am Beispiel von Android sehen wir uns an, welche Einsatzzwecke Rust dort haben könnte.
Stabilität und Sicherheit stehen bei großen Projekten wie Android im Vordergrund. Es muss sehr viel Zeit und Ressourcen in die Behebung von Bugs gesteckt werden, die aus fehlerhaften Speicherzugriffen resultieren. Dies betrifft auch bei Android ungefähr 70% der entdeckten Fehler. [9]
Sprachen zu nutzen, die Speichersicherheit garantieren können und somit diese Fehler gänzlich vermeiden, ist hier die kosteneffektivste Lösung. Wäh-

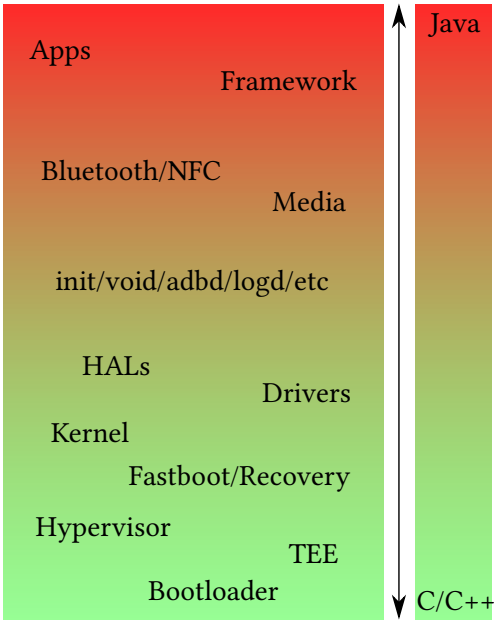

Abbildung 3.1.: Android - Java <-> C/C++ [9]

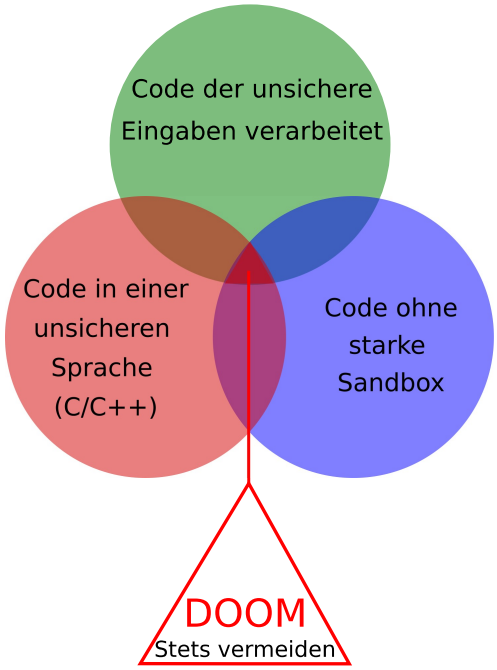

Abbildung 3.2.: Rule of 2 [9]

4. Ausblick

rend große Teile der anwenderzugewandten Seite in Java oder Kotlin implementiert sind, welche durch die Android Runtime gemanaged sind, ist es für alle hardwarenahen Komponenten notwendig, auf C, C++ oder Rust zu setzen. Abbildung 3.1 zeigt die Elemente von Android in ihrer jeweiligen Zuteilung wie nah sie sich am User oder an der Hardware befinden und entsprechend welche Sprache sich für die Implementierung eignet.
Um auch mit unsicheren Sprachen wie C und C++ schwerwiegende Fehler so gut es geht zu vermeiden wird beim Android Projekt die „Rule of 2"- zu deutsch Zweierregel - angewandt. Diese Regel besagt, dass höchstens zwei der Punkte, die wir in Abbildung 3.2 sehen, gleichzeitig zutreffen dürfen. Weil aber Sandboxen viel Ressourcen brauchen und der Code noch immer fehlerhaft sein kann, könnte Rust dazu betragen, nicht nur die Zahl der Bugs zu reduzieren, sondern auch die Effektivität der Sandboxen zu verbessern.
Zudem könnte mehr Code außerhalb von Sandboxen ausgeführt werden, was die Performance verbessert und neue Features ermöglichen könnte.

Systementwicklung: Da Rust vergleichsweise noch eine recht junge Sprache ist in einer recht schwerfälligen Domäne, gibt es noch nicht sehr viele Open Source Projekte, die vollständig auf Rust setzen. Die bisher größten sind Servo [10], eine Browser Engine aus dem Hause Mozilla und der Rust Compiler selbst [11]. Wie wir aber schon am Beispiel von Android gesehen haben, gibt es bei einigen großen Projekten Bestrebungen neuen Code in Rust umzusetzen. Auch bei Chromium wird darüber nachgedacht [3] und Vorgaben zu einer möglichen Integration gemacht [12].
Sogar bei einem der größten Open Source Projekte, dem Linux Kernel, werden aktuell Möglichkeiten geprüft, wie man Rust integrieren kann. [13]. Google ist dafür der Rust for Linux Organisation [14] beigetreten und forscht an der Integration von Rust. Auch Linus Torvalds, Erfinder und Hauptentwickler von Linux, der Veränderungen oft kritisch gegenüber steht und großer Verfechter von C ist, kann sich durchaus vorstellen neue Teile des Kernels zukünftig in Rust zu implementieren. [15] Hauptpotenzial sieht er bei Treibern, da diese unabhängig vom Rest des Kernels agieren und somit prädesti-

niert sind, um eine andere Technologie einzubringen.

Webentwicklung: Doch nicht nur Systemapplikationen können mit Rust umgesetzt werden. Ergebnisse einer Umfrage von 2019 [16] zeigen, dass die meisten Entwickler Rust für Backend Web Applikationen benutzen. Das Gemeinschaftsprojekt „Are we web yet?“[17], zu deutsch „Sind wir bereit fürs Web?“, pflegt eine aktuelle Liste an Rust Technologien, die für Web Entwicklung notwendig sind. Während einige der Projekte noch nicht ganz stabil sind, ist bereits großes Potential erkennbar. Die Sicherheit und Geschwindigkeit von Rust, kombiniert mit der einfachen Bedienbarkeit von Frameworks könnten die Performance von Backend Servern zukünftig deutlich steigern.
Sogar client-seitige Entwicklung mit Rust wird durch die Einführung von WebAssembly möglich. Rust Code wird zu WebAssembly kompiliert und kann direkt im Browser ausgeführt werden. Gerade für Webseiten, bei denen auf Client Seite komplexe Berechnungen ausgeführt werden müssen, könnte Rust bald eine wichtige Rolle spielen.

GUI Entwicklung: Auch native Anwendungen mit graphischer Oberfläche sind mit Rust schon möglich. Dazu wird auf dem Blog „Are we GUI yet?“[17], ähnlich zu „Are we web yet?“, eine Liste gepflegt zu aktuellen Technologien. Bindings zum bekannten GTK existieren bereits und sind schon voll funktionstüchtig [18], während Bindings zu Qt, dem wohl bekanntesten Applikations Framework, noch etwas Entwicklungszeit benötigen. Bisher existiert hier nur ein Projekt, das unsicheren Rust Code aus den C++ Bibliotheken generiert [19]. Wenn hier jedoch die Entwicklung genauso schnell voranschreitet wie in anderen Bereichen, wird es auch hier schon bald mehr Werkzeuge geben.

5. Fazit

Wie wir gesehen haben ist Rust durch seine Eigenschaften durchaus in der Lage, andere systemnahe Sprachen wie C und C++ zu ersetzen. Da die Sprache aber noch recht jung ist, fehlt es wichtigen Bibliotheken teilweise noch an Reife.
Vorteile wie garantiere Nebenläufigkeit und Sicherheit bei Speicherzugriffen stellen aber einen enormen Vorteil gegenüber den bisherigen Sprachen dar. Immer mehr Unternehmen erkennen die Chancen von Rust und sehen die Kosteneinsparungen von wartungsärmerem Code.
Durch die Trägheit von großen Projekten mit Millionen Zeilen an Code wird es jedoch noch einige Zeit dauern bis Rust einen nennenswerten Anteil an neuem Code haben wird.

Literatur

1. *Stackoverflow Survey 2020 Insights* **online**2021 [besucht am 2021-05-29]. Abger. unter: https://insights.stackoverflow.com/survey/2020 (siehe S. 1).
2. TIOBE. *Tiobe Index* **online**2021-05 [besucht am 2021-06-03]. Abger. unter: https://www.tiobe.com/tiobe-index/ (siehe S. 1).
3. *Chromium Security* **online**2021 [besucht am 2021-05-29]. Abger. unter: https://www.chromium.org/Home/chromium-security/memory-safety (siehe S. 3, 6).
4. *Microsoft Blog* **online**2021 [besucht am 2021-05-29]. Abger. unter: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/ (siehe S. 3).

5. *Stackoverflow what is rust and why is it so popular* **online**2020-01-20 [besucht am 2021-05-29]. Abger. unter: https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/ (siehe S. 3, 4).
6. *rustfmt style guide* **online**2021 [besucht am 2021-05-30]. Abger. unter: https://github.com/rust-dev-tools/fmt-rfcs (siehe S. 4).
7. *Rust FAQ* **online**2021 [besucht am 2021-05-29]. Abger. unter: https://prev.rust-lang.org/de-DE/faq.html (siehe S. 3, 4).
8. *Why Go compiles so fast* **online**2021 [besucht am 2021-05-30]. Abger. unter: https://devrajcoder.medium.com/why-go-compiles-so-fast-772435b6bd86 (siehe S. 5).
9. *Rust in Android* **online**2021-04-06 [besucht am 2021-05-29]. Abger. unter: https://security.googleblog.com/2021/04/rust-in-android-platform.html (siehe S. 5, 6).
10. *Servo Github* **online**2021 [besucht am 2021-05-31]. Abger. unter: https://github.com/servo/servo (siehe S. 6).
11. *Rust Ownership* **online**2021 [besucht am 2021-05-29]. Abger. unter: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (siehe S. 4, 6).
12. *Chromium - Rust and C++ interoperability* **online**2021 [besucht am 2021-05-29]. Abger. unter: https://www.chromium.org/Home/chromium-security/memory-safety/rust-and-c-interoperability (siehe S. 6).
13. *Google Security Rust in the Linux Kernel* **online**2021 [besucht am 2021-05-31]. Abger. unter: https://security.googleblog.com/2021/04/rust-in-linux-kernel.html (siehe S. 6).
14. *Github Rust for Linux* **online**2021 [besucht am 2021-05-31]. Abger. unter: https://github.com/Rust-for-Linux (siehe S. 6).
15. ANDREWS, Jeremy. *Tag1 Linus Torvalds Interview* **online**2021-04-28 [besucht am 2021-05-31]. Abger. unter: https://www.tag1consulting.com/blog/interview-linus-torvalds-linux-and-git (siehe S. 6).
16. *Rust Survey 2019 Results* **online**2020-04-17 [besucht am 2021-05-31]. Abger. unter: https://blog.rust-lang.org/2020/04/17/Rust-survey-2019.html (siehe S. 7).
17. *Are we web yet?* **online**2021-05-25 [besucht am 2021-05-31]. Abger. unter: https://www.arewewebyet.org/ (siehe S. 7).
18. *Rust Github* **online**2021 [besucht am 2021-05-31]. Abger. unter: https://github.com/rust-lang/rust (siehe S. 7).
19. *Ritual C++ libraries from Rust* **online**2021 [besucht am 2021-06-01]. Abger. unter: https://rust-qt.github.io/ritual/ (siehe S. 7).