Rustin muistinhallinta ja sen vaikutukset käytännön ohjelmistotuotantoon

Turun yliopisto
Tietotekniikan laitos
TkK-tutkielma
Toukokuu 2025
Eelis Autio

TURUN YLIOPISTO
Tietotekniikan laitos

Eelis Autio: Rustin muistinhallinta ja sen vaikutukset käytännön ohjelmistotuotantoon

TkK-tutkielma, 19 s.

Toukokuu 2025

Muistinhallintaan liittyvät ohjelmointivirheet aiheuttavat merkittäviä ongelmia ohjelmistojen vakauteen ja tietoturvallisuuteen. Näiden ongelmien ratkaisemiseksi on kehitetty muistiturvallisia ohjelmointikieliä, joiden muistiturvallisuus perustuu automaattiseen roskankeruuseen. Rust on ohjelmointikieli, joka takaa ohjelmien muistiturvallisuuden lähdekoodin staattisen analyysin avulla. Tämä mahdollistaa suorituskykyisten ja muistiturvallisten ohjelmistojen kehittämisen. Tämän tutkielman tarkoituksena on selvittää, miten Rust takaa ohjelmien muistiturvallisuuden, ja millaisia haasteita näiden takeiden saavuttaminen aiheuttaa käytännön ohjelmistotuotannossa. Tutkielma on toteutettuna kirjallisuuskatsauksena. Lähdeteokset sisältävät tieteellisiä julkaisuja muistinhallinnasta sekä Rustissa, että muissa ohjelmointikielissä. Lisäksi työssä on viitattu yritysten julkaisemiin artikkeleihin, joissa käsitellään Rustin käyttöä heidän tuotantoympäristössään.

Asiasanat: Rust, muistinhallinta, ohjelmistotuotanto

UNIVERSITY OF TURKU
Department of Computing

Eelis Autio: Rustin muistinhallinta ja sen vaikutukset käytännön ohjelmistotuotantoon

Bachelor's Thesis, 19 p.

May 2025

Memory management errors remain a major source of instability and security vulnerabilities in software systems. To mitigate these issues, memory-safe programming languages have been developed, often relying on automatic garbage collection to ensure safety. Rust is a modern programming language that guarantees memory safety through compile-time static analysis rather than runtime mechanisms. This approach enables the development of high-performance software without sacrificing safety. This thesis aims to examine how Rust enforces memory safety and to explore the practical challenges that arise when adopting Rust in real-world software development. The study is conducted as a literature review, drawing on scientific publications related to memory management in Rust and other programming languages. Additionally, it references industry reports and articles detailing the adoption and use of Rust in production environments.

Keywords: Rust, memory management, software development

Sisällys

1 Johdanto

Ohjelmointikielten lyhyehkön historian aikana on kehitetty lukuisia tekniikoita muistiturvallisten ohjelmien kirjoittamiseen. Laajasti käytössä olevista ohjelmointikielistä (esim. Python, Java, Javascript) valtaosa takaa muistiturvallisuuden siirtämällä muistinhallinnan kehittäjiltä ohjelmointikielen tarjoaman ajoympäristön vastuulle. Näissä kielissä muistinhallinnasta vastaa ajonaikainen roskankeruujärjestelmä (engl. garbage collector, GC), joka toimii seuraamalla ohjelman käyttämää muistia ja vapauttamalla muistialueita automaattisesti, jos se pystyy toteamaan, ettei niille ole enää tarvetta.

Roskankeruu ei sovellu kaikkiin ohjelmistoihin. GC:n ajonaikainen toiminta vaatii tietokoneelta resursseja, joita voidaan tarvita varsinaisen ohjelman ajossa. Perinteisesti suorituskykyä vaativat ohjelmistot, kuten käyttöjärjestelmät, tietokannat, pelimoottorit ja reaaliaikaiset simulaatiot ovat kirjoitettu käyttäen C ja C++ -kieliä. GC:n hyödyntämisen sijaan näissä kielissä muistia hallitaan manuaalisesti, mikä tarkoittaa vastuun muistiturvallisuudesta olevan kehittäjillä.

Microsoftin julkaisemassa artikkelissa [1] arvioidaan, että 70 prosenttia heidän tuotteissaan ja palveluissaan esiintyvistä tietoturvahaavoittuvuuksista johtuu muistinhallinnallisista ohjelmointivirheistä. Kattavin tapa paikata nämä haavoittuvuudet on käyttää muistiturvallisia kieliä, mutta kuten aiemmin todettu, GC:llä muistiturvallisuuden takaavat kielet eivät sovellu kaikkiin ohjelmistoihin.

Vuonna 2015 1.0-julkaisunsa saanut Rust-ohjelmointikieli pyrkii ratkaisemaan

ongelman, jossa valinta käytettävästä ohjelmointikielestä tarkoittaa valintaa muistiturvallisuuden ja nopeuden väliltä. Rust on C/C++:an tavoin nk. järjestelmäkieli (eng. system programming language), mutta niistä poiketen muistiturvallinen. Ajonaikaisen järjestelmän sijaan Rust antaa kehittäjälle tietyt takeet ohjelman muistiturvallisuudesta käännösaikana tehtyjen tarkastusten avulla (staattinen analyysi). Nämä tarkastukset muodostavat laajan ja monivaiheisen prosessin, jonka tarkastelu ei kuulu tämän tutkielman piiriin. Staattinen analyysi kuitenkin perustuu pitkälti Rustin tyyppijärjestelmään, jonka tarkastelu on olennaista Rustin käytettävyyden arvioinnissa käytännön ohjelmistotuotannon näkökulmasta. Tästä syystä tutkielmaan sisältyy pintapuolinen käsittely Rustin tyyppijärjestelmästä. Varsinaisia tutkimuskysymyksiä ovat:

1. Miten Rust takaa ohjelmien muistiturvallisuuden?

2. Millaisia haasteita Rustin muistiturvallisuuden takaava mekanismi aiheuttaa käytännön ohjelmistotuotannossa?

1.1 Rakenne

Tämän tutkielma koostuu kahdesta pääosasta. Taustaluvussa käsitellään ensimmäinen tutkimuskysymys, johon sisältyy Rustin tyyppijärjestelmä, omistus- ja lainaussäännöt sekä katsaus siitä, miten nämä konseptit mahdollistavat muistiturvallisten ohjelmien kirjoittamisen ilman GC:tä. Taustaluvun tarkoitus on antaa lukijalle kuva siitä, millaisten ominaisuuksien varaan kielen muistiturvallisuus perustuu.

Taustaluvusta saatu ymmärrys auttaa lukijaa arvioimaan toisen käsittelyluvun sisältöä, jonka tarkoituksena on arvioida Rustiin liittyviä haasteita käytännön ohjelmistotuotannossa. Toinen tutkimuskysymys on muotoiltu kiinnittämään huomiota nimenomaan haasteisiin, sillä muistiturvallisuuden hyötyjen perustelu ei ole tämän tutkielman tarkoitus. Niiden puolesta puhuu muistiturvallisten ohjelmointikielten

ylivoimainen suosio nykyiaikaisessa ohjelmistotuotannossa [2].

2 Taustaa

Rust-kielen verkkosivuilla tärkeimpinä syinä kielen käyttämiseen mainitaan suorituskyky, luotettavuus ja tuottavuus (tarkistettu 17.11.2024). Näistä attribuuteista luotettavuuteen sisältyy kielen takeet muistiturvallisuudesta. Tämän luotettavuuden saavuttamiseksi kielessä on yhdistetty useita eri tekniikoita ja ideoita tyyppijärjestelmistä ja resurssienhallinnasta [3]. Esimerkkeinä näistä mainittakoon C++ -kieleen alunperin kehitettyä RAII-resurssienhallinta (engl. resource acquisition is initialization), jossa objektin käyttämä resurssi hankitaan sen konstruktorissa ja vapautetaan destruktorissa [4]. RAII-malli siis sitoo resurssien eliniän sitä käyttävän objektin elinikään. Rust ohjaa kehittäjiä noudattamaan tätä mallia automatisoimalla. Siinä missä C++ jättää kehittäjän vastuulle huolehtia resurssien vapauttamisesta destruktoreissa, tekee Rust tämän automaattisesti tilanteissa, joissa se on mahdollista.

2.1 Omistus- ja lainaussäännöt

Omistus- ja lainaussäännöt ovat Rustin muistiturvallisuuden perusta [5]. Nämä säännöt määrittelevät tavat, joilla ohjelman on sallittua käyttää muistiaan ajon aikana. Niiden noudattamista valvoo Rustin kääntäjä *rustc*, joka hylkää ohjelman virheellisenä, mikäli lähdekoodi ei noudata omistus- ja lainaussääntöjä. Näistä säännöistä voidaan tarvittaessa joustaa Rustin `unsafe`-avainsanan avulla, jota käytettäessä muistiturvallisuudesta vastaa rustc:n sijaan kehittäjä itse. Koska tarkoituksena on

käsitellä nimenomaan turvallista Rustia, ei `unsafe`-avainsanan käsittely kuulu tähän tutkielmaan.

2.1.1 Omistussäännöt

Omistussäännöt, kuten ne on Rust-kirjassa esitelty:

1. "Each value in Rust has an owner."
2. "There can only be one owner at a time."
3. "When the owner goes out of scope, the value will be dropped"

Ensimmäinen omistussääntö

Ensimmäinen sääntö esittelee omistuksen konseptin. Rustissa omistajalla tarkoitetaan muuttujaa tai vakiota, joka "omistaa"arvonsa. Esimerkkinä ohjelmalistauksessa 1 määritellään muuttuja `s` ja sen omistama arvo, joka on tyypiltään `String`.

Ohjelmalistaus 1 Muuttuja `s`.

```
let s: String = String::from("Esimerkki");
```

Toinen omistussääntö

Toinen sääntö kertoo, että arvolla voi olla kerrallaan vain yksi omistaja. Tätä voidaan demonstroida luomalla uusi muuttuja `s2`, asettamalla sen arvoksi `s` ja yrittämällä tulostaa `s`:n arvo:

Ohjelmalistaus 2 Omistajuuden siirtäminen

```
let s2 = s;
println!("{}", s);
```

Ohjelmaa käännetäessä rustc antaa seuraavan virheilmoituksen:

```
error[E0382]: borrow of moved value: `s`
--> src/main.rs:7:20
|
5 | let s: String = String::from("Esimerkki");
| - move occurs because `s` has type `String`, which does not implement
the `Copy` trait
6 | let s2 = s;
| - value moved here
7 | println!("{}", s);
| ^ value borrowed here after move
```

Tämä virheilmoitus kertoo, että muuttujan `s` arvon omistajuus on siirtynyt muuttujalle `s2`, eikä sitä voida tämän jälkeen käyttää. Virheilmoitus johtuu siis toisen omistussäännön noudattamatta jättämisestä. On syytä huomioida omistajuuden siirtymisen johtuvan siitä, ettei String-tyyppi implementoi Copy-rajapintaa. Tämä rajapinta voidaan toteuttaa vain sellaisille tyypeille, joiden koko tiedetään kääntöaikana. String-tyypin sisältämä teksti säilytetään ohjelman dynaamisessa muistissa, joten sen koko voi muuttua ajon aikana [6].

Kolmas omistussääntö

Kolmas omistussääntö kertoo arvon käyttämän muistin vapautuvan sen omistajan poistuessa näkyvyysalueelta. Tätä voidaan demonstroida luomalla uusi näkyvyysalue ja uusi muuttuja sen sisälle, ja yrittämällä tulostaa arvo näkyvyysalueen ulkopuolella:

Ohjelmalistaus 3 Vapautettuun muistiin viittaaminen

```
{
let s: String = String::from("Esimerkki");
}
println!("{}", s);
```

Tämä koodi aiheuttaa seuraavan virheilmoituksen:

```
error[E0425]: cannot find value `s` in this scope
--> src/main.rs:8:20
|
8 | println!("{}", s);
|
```

Virheilmoituksesta selviää, ettei muuttujaa `s` löytynyt näkyvyysalueelta, jossa sitä yritettiin käyttää. Rustissa muuttujan näkyvyysalue alkaa kohdasta, jossa se on luotu, ja päättyy sen näkyvyysalueen loppuun, jossa luonti tapahtui [6]. Näkyvyysalueita merkitään C/C++:an tavoin aaltosulkeilla. Tyypillisiä näkyvyysalueita ovat funktioiden ja silmukoiden rungot.

2.1.2 Lainaussäännöt

Arvojen liikuttelu näkyvyysalueiden välillä ja omistussääntöjen puitteissa voi osoittautua hankalaksi monimutkaisemmissa ohjelmissa. Tätä varten Rustissa on omistettujen tyyppien lisäksi lainattavia tyyppejä, jotka eivät omista viittaamaansa dataa [Zeeb 2022]. Rustissa näitä tyyppejä kutsutaan viittauksiksi (engl. reference). Viittaukset ovat C/C++:an tapaan käytännössä toteutettu osoittimina (engl. pointer), sillä ne sisältävät muistiosoitteen, josta viitattu data löytyy. C ja C++ kielistä poiketen Rust takaa viittausten osoittavan aina validiin arvoon viittauksen näkyvyysalueella [Rust book]. Viittausten validiuden takaamiseksi rustc valvoo lainaussääntöjen noudattamista, joiden puitteissa kaikkien lainausten on tapahduttava. Rust book esittää lainaussäännöt seuraavasti:

1. "At any given time, you can have either one mutable reference or any number of immutable references."

2. "References must always be valid."

Ensimmäinen lainaussääntö

Ensimmäisessä lainaussäännössä puhutaan muuttuvista ja muuttumattomista viittauksista. Muuttuvia viittauksia voi olla kerrallaan vain yksi. Vaihtoehtoisesti muuttumattomia viittauksia voi olla useita, muttei samaan aikaan muuttuvien viittausten kanssa. Viittauksen muuttuvuudella tarkoitetaan lainaajan oikeutta muokata viitattua dataa. Esimerkiksi funktio, jonka parametreissä on määritelty sen ottavan vastaan muuttuva viittaus dynaamiseen listaan, voi näkyvyysalueellaan muokata tämän listan sisältöä.

Demonstroidaan ensimmäistä lainaussääntöä esimerkillä:

Ohjelmalistaus 4 Viittausten luominen

```
let s = String::from("Esimerkki");
let s2 = &s;
let s3 = &s;
println!("{}, {}, {}", s, s2, s3);
```

Ohjelma kääntyy virheettä ja tulostaa: "Esimerkki, Esimerkki, Esimerkki". Koodi noudattaa lainaussääntöjä, sillä `s2` ja `s3` ovat muuttumattomia viittauksia, eli niillä ei ole oikeutta muokata lainattua dataa. Koodia voidaan muokata tekemällä `s2`:sta muuttuva lainaus:

Ohjelmalistaus 5 Muuttuvan lainaus `s2`

```
let mut s = String::from("Esimerkki");
let s2 = &mut s;
let s3 = &s;
println!("{}, {}, {}", s, s2, s3);
```

Nyt koodin kääntäminen aiheuttaa seuraavan virheilmoituksen:

```
error[E0502]: cannot borrow `s` as immutable because it is also
borrowed as mutable
--> src/main.rs:8:14
|
7 | let s2 = &mut s;
| ------ mutable borrow occurs here
8 | let s3 = &s;
| ^^ immutable borrow occurs here
9 |
10 | println!("{}, {}, {}", s, s2, s3);
| -- mutable borrow later used here
```

Virheilmoituksesta voi ymmärtää virheen johtuneen ensimmäisen lainaussäännön rikkomisesta. Muuttujan `s` omistamaa dataa ei voida lainata muuttumattomana `s3`:lle, sillä se on jo lainattu muuttuvana `s2`:lle. Ensimmäinen lainaussääntö estää sellaisten tilanteiden syntymisen, joissa kaksi säiettä yrittävät muokata jonkin muuttujan arvoa samanaikaisesti (engl. data race) [6].

Toinen lainaussääntö

Toinen lainaussääntö vaatii kaikkien viittausten olevan valideja. Validi viittaus tarkoittaa viitattavan datan olevan oikeaa tyyppiä ja olemassa. Rust sallii alustamattomien muuttujien luomisen, mutta niihin viittaminen aiheuttaa käännösaikana virheen, sillä se rikkoisi toista lainaussääntöä. Voimme demonstroida tätä seuraavalla koodipätkällä:

Ohjelmalistaus 6 Virheellinen viittaus `s2`

```
let s: String;
let s2 = &s;
```

Koodin kääntäminen aiheuttaa seuraavan virheen:

```
error[E0381]: used binding `s` isn't initialized
--> src/main.rs:7:14
|
6 |     let s: String;
|         - binding declared here but left uninitialized
7 |     let s2 = &s;
|              ^^ `s` used here but it isn't initialized
```

Virhe kertoo, ettei muuttujaa `s` ole alustettu eikä sitä siksi voida lainata muuttujalle `s2`.

Toinen lainaussääntö tarkoittaa myös, että lainatut arvot täytyy palauttaa eheinä. Funktio, joka ottaa syötteenään muuttuvan viittuaksen, ei voi muuttaa viitatun arvon tyyppiä tai vapauttaa sen käyttämää muistia. Seuraava esimerkki demonstroi tilannetta, jossa funktio yrittää asettaa uuden arvon sille parametrina annetulle muuttuvalle viittaukselle.

Ohjelmalistaus 7 Roikkuva viittaus

```
fn main() {
    let mut s = String::from("Esimerkki");
    reassign(&mut s);
}

fn reassign(mut s: &mut String) {
    let mut s2 = String::from("Uusi arvo");
    s = &mut s2;
}
```

Koodin kääntäminen aiheuttaa seuraavan virheilmoituksen:

```
error[E0597]: `s2` does not live long enough
--> src/main.rs:11:9
|
9  | fn reassign(mut s: &mut String) {
|                    - let's call the lifetime of this reference `'1`
10 |     let mut s2 = String::from("Uusi arvo");
|         ------ binding `s2` declared here
11 |     s = &mut s2;
|     ----^^^^^^^
|     |   |
|     |   borrowed value does not live long enough
|     assignment requires that `s2` is borrowed for `'1`
12 | }
| - `s2` dropped here while still borrowed
```

Ohjelman reassign-funktiossa luodaan uusi String-tyyppinen muuttuja `s2` ja yritetään siirtää funktion parametrina annettu viittaus `s` osoittamaan tähän. Virheilmoitus on seuraus kolmannesta omistussäännöstä. Koska `s2` on määritelty vain funktion näkyvyysalueella, vapautetaan sen käyttämä muisti funktion lopussa. Funktion suorittamisen jälkeen `s` viittaisi vapautettuun muistiin, mikä on toisen lainnaussäännön vastaista.

2.2 Tyyppijärjestelmä

Rustc:n tekemä staattinen analyysi perustuu pitkälti kielen tyyppijärjestelmään, jonka tarkastelu on olennaista Rustin arvioinnissa käytännön ohjelmistotuotannon näkökulmasta. Rust on staattisesti tyypitetty kieli, mikä tarkoittaa muuttujien olevan kääntöaikana jotain määriteltyä tyyppiä [Zeeb 2022]. Rustin tyypit voidaan kategorisoida useilla tavoilla, mutta yhdistävänä tekijänä on määriteltävyys. Tämä tarkoittaa, että jokaisella validilla tyypillä on jokin määriteltävissä oleva arvojoukko [RustBelt 2018]. Tämän luvun tarkoituksena on tarkastella Rustin tyyppijärjestelmän merkittävimpiä osia sekä tapoja saavuttaa sellaisia C-kielistä tuttuja toiminnallisuuksia, joita omistus- ja lainaussäännöt pintapuolisesti näyttäisivät estävän.

2.2.1 Eliniät

Eliniät (engl. lifetimes) erottavat Rust-lainaukset C-kielissä käytettyistä osoittimista. Siinä missä C:n osoittimen `T* ptr`-tyypistä saadaan selville ainoastaan osoitetun arvon tyyppi `T`, Rust lainaus `&'a T` sisältää myös eliniän `'a`, jonka avulla rustc kykenee validoimaan lainauksen [3]. Kääntäjä pystyy usein päättelemään lainausten eliniät ja lisäämään ne koodiin kääntövaiheessa, jolloin käyttäjän ei tarvitse niitä erikseen merkitä. Jokaisella lainauksella elinikä on kuitenkin implisiittisesti olemassa.

Seuraavassa esimerkissä havainnolistetaan tilannetta, jossa tietueen `Ihminen` sisältämä kenttä `nimi` on viittaus String-tyyppiin. `Ihminen`-tyyppiset muuttujat ovat siis riippuvaisia rakenteen ulkopuolella sijaitsevasta datasta:

```
struct Ihminen {
    nimi: &String,
    ikä: u8,
}

fn main() {
    let nimi_muuttuja: String = "Esimerkki".to_string();
    let henkilö: Ihminen = Ihminen { nimi: &nimi_muuttuja, ikä: 20 };
    println!("{}", henkilö.nimi);
}
```

Kääntäjä antaa yo. koodista seuraavan virheilmoituksen:

```
error[E0106]: missing lifetime specifier
--> src/main.rs:2:11
|
2 |     nimi: &String,
|           ^ expected named lifetime parameter
|
help: consider introducing a named lifetime parameter
|
1 ~ struct Ihminen<'a> {
2 ~     nimi: &'a String,
|
```

Esimerkissä muuttuja `henkilö` on riippuvainen muuttujasta `nimi_muuttuja`. Viittauskenttiä sisältävät tietueet ovat Rustissa sallittuja, mutta vaativat eksplisiittisen elinikäannotaation käyttöä. Korjataan esimerkin ohjelma toimivaksi:

```
struct Ihminen<'a> {
    nimi: &'a String,
    ikä: u8,
}
...
```

Tämän muutoksen jälkeen koodi kääntyy ongelmitta. Syy tähän ei kuitenkaan ole ilmeinen. Alkuperäistä koodia lukemalla voidaan todeta ohjelman noudattavan lainaussääntöjä, sillä muuttujan `henkilö` lainaama muuttuja `esimerkki_nimi` on validi läpi ohjelman ajon. Tehdäkseen tämän päätelmän kääntäjä tarvitsee kuitenkin enemmän tietoa. Annotaatio `'a` linkittää tietueen eliniän sen lainaaman muuttujan elinikään siten, että kääntäjä voi varmistaa lainattavan muuttujan eliniän olevan vähintään yhtä pitkä, kuin lainaavalla rakenteella [7]. Esimerkissä `henkilö` ei siis voi elää `nimi_muuttuja`a pidempään, sillä jos näin olisi, voisi henkilö-tietueen kautta mahdollisesti viitata vapautettuun muistiin.

3 Rust käytännön ohjelmistotuotannossa

Kuten ylempänä todetaan, Rust tarjoaa enemmän takeita muistiturvallisuudesta, kuin monet muut ohjelmointikielet. Kielen käytettävyyden arvioinnissa käytännön ohjelmistotuotannossa on kuitenkin syytä tarkastella millaisella hinnalla kielen edut saavutetaan. Tutkimuksessa [8] selvitettiin Stack Overflow -palvelun tarjoaman datan ja tutkijoiden teettämän kyselytutkimuksen avulla Rust-kehittäjien ymmärrystä kielen turvallisuutta takaavista säännöistä. Tutkimuksessa todettiin sääntöjen olevan monimutkaisia käytännössä. Sääntöjen teoreettinen ymmärtäminen ei ollut tae kyvystä ymmärtää virheiden juurisyitä käytännön ohjelmissa. Tutkimuksessa todettiin myös viittausten elinikäsääntöjen olevan omistus- ja lainaussääntöjä suurempi kipupiste kielen oppimisessa.

Toinen Rustin etuja ja haasteita käytännön ohjelmistotuotannossa tarkasteleva tutkimus [7] tunnistaa joitakin haasteita kielen käyttöönotossa ohjelmistotaloissa ja -osastoilla. Haasteet jaettiin kahteen kategoriaan: yleiset uusien ohjelmointikielen käyttöönottoon liittyvät haasteet ja nimenomaisesti Rustiin liittyvät haasteet. Yleisiksi haasteiksi mainittiin esimerkiksi yhteensopivuusongelmat olemassaolevan koodin kanssa ja halu välttää ylimääräisiä muutoksia. Rustiin liittyviksi haasteiksi tunnistettiin kielen jyrkkä oppimiskäyrä ja sen nuoruudesta johtuvat puutteet, kuten ekosysteemin kattamattomuus ja Rust-kehittäjien palkkaamisen vaikeus. Rust

on nuori kieli, jonka 1.0-versio julkaistiin toukokuussa 2015. Kielen suosio on ollut jatkuvassa nousussa julkaisuustaan asti; 2019 se esiintyi ensimmäistä kertaa Stack Overflown vuosittain teettämässä kyselyssä suosituimpien ohjelmointikielien listalla. Kielen suosio ei kuitenkaan yllä sen suurimman kilpailijan, eli C++:an tasolle, mikä osaltaan selittää Rust-kehittäjien palkkaamiseen liittyviä vaikeuksia. Käyttöönottoa hidastavista kolmesta nimenomaisesti Rustiin liittyvästä päätekijästä kaksi liittyvät kielen nuoruuteen, eivätkä siksi ole esteitä Rustin käyttöönotolle pidemällä aikajanalla.

Rustin positiivisista vaikutuksista käytännön kehitystyöhön tutkimuksessa [7] mainitaan koodin luotettavuus, kehityssyklin tuottavuuden kasvu ja kehittäjien parantunut tietoisuus turvallisuudesta muissa kielissä. Tutkimuksessa teetetyn kyselyn vastauksista ilmenee kääntäjän tekemän staattisen analyysin hyöty: mikäli koodi kääntyy, voi kehittäjä olla suhteellisen varma sen oikeellisuudesta. Tästä seuraa, että virheenjäljitykseen (debuggaus) käytetty aika on vähäistä verrattuna kieliin, joissa kehittäjällä on suurempi vastuu muistinhallinnan oikeellisuudesta. Vastaajien keskuudessa vallitsi yhteisymmärrys siitä, että vaikka ohjelmiston alustava kehitys kesti Rustissa pidempään, oli projektiin kokonaisuudessa käytetty aika lyhempi, sillä virheenjäljityksessä säästetty aika riitti kumoamaan alkuvaiheessa ohjelmointiin käytetyn pidentyneen ajan.

Google Open Source -blogin kirjoituksessa [9] käsitellään viittä Rustin käyttöönottoon Googlella liittyvää havaintoa. Blogikirjoituksen perustana toiminut yhtiön sisäinen kysely kartoitti yli tuhannen Rust-koodia työssään kirjoittaneen kehittäjän kokemuksia kielestä ja sen oppimisesta. Vastaajista 39,8 prosenttia kertoi kielen opiskelun sellaisele tasolle, jolla ohjelmoija luotti kykyynsä kirjoittaa Rustia, kestäneen yhdestä kahteen kuukautta. Toiseksi suurin ryhmä (27 prosenttia) kertoi tämän tason saavuttamisen kestäneen vain kahdesta kolmeen viikkoa. Suurimmiksi haasteiksi kielen oppimisessa kirjoitus mainitsee makrot, omistus- ja lainaussäännöt, sekä

async-ohjelmoinnin. Kirjoittajat raportoivat, etteivät havainneet mitään indikaatioita Rustiin siirtymisestä johtuvasta kehittäjien tuottavuuden laskusta. Tuottavuuden noususta ei kirjoituksessa ole mainintaa. Sen sijaan koodin laadun paranemisesta löytyi näyttöä: kyselyssä 85 prosenttia vastaajista koki kirjoittamansa Rust-koodin olevan muilla kielillä kirjoittamaansa koodia oikeellisempaa.

Kehitystyön vaativuutta Rustissa verrattuna C-kieleen arvioidaan tutkimuksessa [10]. Tutkimuksessa toteutettiin gravitaatiosimulaatio molemmissa kielissä ja arvioitiin sekä kehitystyön haastavuutta, että toteutuksien suoritusnopeutta. Uusien simulaatioaskelten suorittamiksesi laskettiin simuloitujen kappaleiden keskenäiset vuorovaikutukset. Täten käytetty algoritmi oli aikavaativuudeltaan $O(n^2)$. Algoritmin naiivi implementaatio on kielestä riippumatta yksinkertainen:

```
FOR every body i = 1 to N
    FOR every body j = 1 to N
        Lasketaan voima, jolla kappale j vaikuttaa kappaleeseen i.
        Lisätään tulos voimien summaan.
    Lasketaan kappaleen i siirtymä
    Liikutetaan kappaletta i
```

Tutkimuksessa luotiin optimoidut versiot tästä algoritmista C- ja Rust-kielellä. Rust-koodin tärkeimmät optimoinnit olivat säikeistäminen, vektorisaatio SIMD-operaatioita varten, sekä matemaattiset optimoinnit, jotka lyhensivät suoritusaikaa tarkkuuden kustannuksella. C-koodin merkittävimmät optimoinnit perustuivat myös säikeistämiseen, SIMD-operaatioihin ja matemaattisiin optimointeihin. Suoritusnopeuksien vertailuissa ei havaittu merkittäviä eroja kielten välillä.

Tutkijat arvioivat toteutusten kehittämiseen käytettyä työpanosta. He tunnistivat kehitystyötä helpottaneita seikkoja molemmista kielistä. C:n eduksi tunnistettiin matemaattisten optimointien helppous. Ne pystyttiin toteuttamaan kääntäjille annettujen ohjeiden avulla. Rustin eduksi tunnistettiin kielen modernit ominaisuudet, kuten lainaukset funktionaalisesta- ja olio-ohjelmoinnista, joiden koettiin lisänneen

ergonomiaa ja koodin ymmärrettävyyttä. Rustin eduksi katsottiin myös ohjelmakirjastojen hallintajärjestelmä Cargo, jonka kautta ladatut kirjastot yksinkertaistivat säikeistämisprosessia huomattavasti.

On huomattava kehitystyön keskittyneen laajan ohjelmiston sijaan algoritmin optimointiin. Tässä tapauksessa Rustin muistinhallintaan liittyvät säännöt eivät muodostuneet ongelmaksi, sillä käsitelty tieto oli keskitetty yhteen tietueeseen, jolloin omistus- ja lainaussääntöjen noudattaminen on suoraviivaista. Tilanne ei kuitenkaan olisi yhtä yksinkertainen, mikäli optimoitava algoritmi käsittelisi syklisiä tietorakenteita.

Rustin ensimmäinen omistussääntö kertoo, että jokaisella arvolla on omistaja. Tämä on ongelmallista tilanteissa, joissa mallinnettavalla datalla ei ole luonnollisia omistajuussuhteita. Esimerkkinä tästä toimii sosiaalisen median palvelu, jossa käyttäjien kaverisuhteita mallinnetaan verkkona. Käyttäjää mallintavalla tietueella tulee olla jokin tapa pitää kirjaa muista käyttäjistä, jotka ovat tämän kaverilistalla. Yksinkertaisin tapa toteuttaa tämä olisi kenttä käyttäjätietueessa, joka pitäisi listaa viittauksista muihin käyttäjiin. Turvallisessa Rustissa tämä ei ole mahdollista, sillä lainaussäännöt takaavat viittausten olevan valideja, mutta käyttäjätilin poistaminen johtaisi kaikkien siihen viitanneiden käyttäjien pitävän invalidia viittausta omassa kaverilistassaan.

Syklisiä tietorakenteita kuitenkin tarvitaan. Paljaat osoittimet (engl. raw pointer) ovat yksi tapa mallintaa niitä, mutta ne vaativat unsafe-avainsanan käyttöä. Toinen vaihtoehto syklisten viittausten toteuttamiseksi on käyttää standardikirjaston tarjoamia älykkäitä osoitintyyppejä. Nämä tyypit pitävät sisällään viittauksen lisäksi metadataa, jonka avulla viittauksen validius voidaan varmistaa [5]. Älykkäiden osoittimien käyttö kuitenkin hidastaa ohjelmaa ajon aikana, sillä lainauksen validioimiseksi tehävä työ tapahtuu käännösajan sijasta ohjelman suorituksen aikana.

Kolmas tapa mallintaa syklisiä tietorakenteita on kiertää Rustin tekemät tarkistukset viittausten validiudesta kokonaan pitämällä käyttäjätietueessa viittausten sijaan listaa id-kentistä, joilla käyttäjät voidaan yksilöidä. Käyttäjätietue voisi näyttää siis tältä:

```
struct User {
    id: u32,
    friends: Vec<u32>
}
```

Tässä tietueessa **User** omistaa kaiken pitämänsä datan, joten lainaussääntöjä ei rikota. Tietue sisältää kuitenkin vain tiedon yhdestä käyttäjästä, eikä sen kautta päästä käsiksi muihin käyttäjäverkon jäseniin. Tätä varten tarvittaisiin toinen tietue; kuten lista tai hajautustaulu, joka omistaisi kaikki käyttäjätilit, ja jonka kautta muihiin käyttäjätileihin päästäisiin käsiksi.

4 Päätelmiä

Tässä tutkielmassa esitettyjen havaintojen perusteella voidaan arvioida Rustin käyttöönoton hyötyjä ja haasteita käytännön ohjelmistokehityksessä. Tarkasteltujen aineistojen perusteella Rustin suurin hyöty on koodin luotettavuudessa. Kääntäjän tekemä staattinen analyysi estää useiden virhekategorioiden esiintymisen Rust-ohjelmissa, mikä on kiistaton hyöty ammattimaisissa ympäristöissä, joissa palveluiden luotettavuus on suorassa yhteydessä asiakastyytyväisyyteen ja ylläpitokuluihin.

Käyttöönoton suurimmat haasteet liittyvät perimmiltään osaamisen puutteeseen. Kasvavasta huomiostaan huolimatta Rust ei tällä hetkellä yllä Stack Overflown vuosittain tekemässä kyselytutkimuksessa käytetyimpien ohjelmointikielien listalla kymmenen suosituimman kielen joukkoon. Rustin käyttöä harkitseville yrityksille vaihtoehdoiksi jää henkilöstön koulutus tai rekrytointi niukasta Rust-osaajien joukosta.

Kielen asettamaksi haasteeksi todettakoon myös, että koodin turvallisuusvaatimuksien täyttymisen todistamiseen käytetty staattinen analyysi ei kykene todistamaan kaikkia oikeellisia ohjelmia turvallisiksi. Tästä parhaana esimerkkinä toimii sykliset tietorakenteet, joiden toteuttaminen turvallisessa Rustissa vaatii joko älykkäiden osoitintyyppien käyttöä, tai viittausten korvaamista indekseillä. Molemmissa tapauksissa toteutuksella on C/C++:aan verrattuna ylimääräinen hinta joko ajonaikaisen suorituskyvyn tai koodin ymmärrettävyyden kannalta.

Lähdeluettelo

[1] G. Thomas, ”A proactive approach to more secure code”, *Microsoft Security Response Center Blog*, 2019.

[2] ”2024 Developer Survey”, 2024. url: `https://survey.stackoverflow.co/2024/technology/#most-popular-technologies`.

[3] K. Ferdowsi, ”The usability of advanced type systems: Rust as a case study”, tammikuu 2023. arXiv: `2301.02308 [cs.PL]`.

[4] G. D. R. Bjarne Stroustrup Herb Sutter, ”A brief introduction to C++’s model for type- and resource-safety”, 2015. url: `https://www.stroustrup.com/resource-model.pdf`.

[5] ”The Rust Programming Language”, 2024. url: `https://doc.rust-lang.org/stable/book/`.

[6] C. Zeeb, ”Memory Management In Rust. Principles and Comparison with C and C++”, 2022. url: `https://www.en.pms.ifi.lmu.de/publications/projektarbeiten/Claudio.Zeeb/BA_Claudio.Zeeb.pdf`.

[7] K. R. Fulton, A. Chan, D. Votipka, M. Hicks ja M. L. Mazurek, ”Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study”, teoksessa *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, USENIX Association, elokuu 2021, s. 597–616, ISBN: 978-

1-939133-25-0. url: https://www.usenix.org/conference/soups2021/presentation/fulton.

[8] S. Zhu, Z. Zhang, B. Qin, A. Xiong ja L. Song, ”Learning and programming challenges of rust: a mixed-methods study”, teoksessa *Proceedings of the 44th International Conference on Software Engineering*, sarja ICSE ’22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, s. 1269–1281, ISBN: 9781450392211. DOI: 10.1145/3510003.3510164. url: https://doi.org/10.1145/3510003.3510164.

[9] K. B. Lars Bergstrom, ”Rust fact vs. fiction: 5 Insights from Google’s Rust journey in 2022”, *Google Open Source Blog*, 2023.

[10] M. Costanzo, E. Rucci, M. Naiouf ja A. D. Giusti, *Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body*, 2021. arXiv: 2107.11912 [cs.PL]. url: https://arxiv.org/abs/2107.11912.