Juuso Lahtinen

Muistiturvallinen ohjelmistokehitys Rustilla

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

11.04.2024

Tiivistelmä

Tekijä:	Juuso Lahtinen
Otsikko:	Muistiturvallinen ohjelmistokehitys Rustilla
Sivumäärä:	31 sivua
Aika:	11.04.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikan tutkinto-ohjelma
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaaja:	Lehtori Simo Silander

Opinnäytetyön tarkoituksena on perehtyä muistiturvallisuuteen, sen käyttötarpeeseen ja hyötyyn. Muistiturvallisuuden avuksi työssä perehdytään muistiin ja muistinhallintaan. Myös ohjelmointikieliin tutustutaan, koska niiden avulla pystytään vertailemaan kielten välisiä eroja sekä havainnollistamaan muistiturvallisuuden tarvetta. Ohjelmointikielet hallitsevat muistia eri tavoin, mutta yleisesti niissä käytetään roskankeräystä muistinhallintaan tai manuaalista muistinhallintaa.

Rust poikkeaa tästä siten, että Rust ei käytä roskankeräystä, mutta ei myöskään manuaalista muistinhallintaa. Opinnäytetyössä syvennytään Rustiin tarkemmin. Tutustutaan itse ohjelmointikielen teoriaan sekä luodaan sovellus havainnollistamaan Rustin toimintoja ja muistinhallintaa. Vertailun vuoksi sama sovellus luodaan C-ohjelmointikielellä.

Muistiturvallisuus koskee myös tietoturvaa, jota opinnäytetyössä tarkastellaan. Tutustutaan muutamiin haavoittuvuuksiin ja tapoihin, miten niitä voidaan estää esimerkiksi eri työskentelymenetelmien avulla.

Opinnäytetyön tuloksena lukija ymmärtää työssä käsitellyt aiheet. Rustilla ja C:llä saatiin onnistuneesti luotua sovellus, joka demonstroi ohjelmointikielten välisiä eroja muistiturvallisuuden saavuttamiseksi.

Avainsanat: Rust, muistiturvallisuus, ohjelmistokehitys

Abstract

Author:	Juuso Lahtinen
Title:	Memory Safe Development with Rust
Number of Pages:	31 pages
Date:	11 April 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisor:	Simo Silander, Senior Lecturer

The purpose of this thesis is to introduce memory safety, its need, and its benefits. To help understand memory safety, the thesis delves into memory and memory management. Programming languages are also explored because they help in comparing differences between languages and illustrating the need for memory safety. All programming languages manage memory differently, but generally, they use garbage collection or manual memory management.

Rust deviates from the above, in that Rust does not use garbage collection, but also does not use manual memory management. The thesis dives deeper into Rust. The theory of the Rust programming language itself is examined, and an application was created to showcase Rust's functions and memory management. For comparison, the same application was created using C programming language.

Memory safety also pertains to information security, which is examined here. Several vulnerabilities and ways to prevent them, for example through different working methods, are explored.

A successful application was created with Rust and C to demonstrate the differences between programming languages and to achieve memory safety.

Keywords: Rust, memory safety, development

Sisällys

Lyhenteet

HDD: *Hard Disk Drive.* Kiintolevyasema, joka on tietokoneeseen asennettu levymuisti, johon tallennetaan eri tiedostot ja ohjelmat. Tallennettu data säilyy tietokoneen sammuessa.

LIFO: *Last In, First Out.* Varastointiperiaate, jossa varastoon viimeiseksi jätetty tavara lähtee varastosta ensimmäisenä.

Mutex: *Mutual exclusion.* Synkronointiperiaate, jota hyödynnetään samanaikaisessa ohjelmoinnissa. Tällä estetään samanaikainen pääsy säikeiltä jaettuun tilaan.

NSA: *National Security Agency.* Yhdysvaltain tiedusteluelin ja -virasto.

RAM: *Random Access Memory.* Tietokoneen muisti, johon data tallennetaan, jota tietokoneen prosessori tarvitsee esimerkiksi sovellusten suorittamiseen ja tiedostojen avaamiseen. Data ei säily tietokoneen sammuessa.

SQL: *Structured Query Language*. Standardoitu ohjelmointikieli, jota käytetään tietokantojen hallintaan.

SSD: *Solid-state Drive.* SSD-levy, myös tietokoneeseen asennettu levymuisti, johon tallennetaan eri tiedostot ja ohjelmat. SSD-levyt käyt-tävät flash-muistia tallennukseen, mikä johtaa parempaan suorituskykyyn. Tallennettu data säilyy tietokoneen sammuessa.

VoIP: *Voice Over Internet Protocol*. Teknologia, joka mahdollistaa reaaliaikaisen puhekeskustelun internetin välityksellä.

1 Johdanto

Ohjelmistokehitys on iso osa nykypäivää, oli se sitten työssä tai arjessa. Ohjelmistoja kehitetään jatkuvasti lisää ja vanhoja päivitetään. Nykypäivän standardeissa ohjelmistojen oletetaan täyttävän tietyt vaatimukset. Näitä ovat esimerkiksi ohjelmiston yleinen toiminta, saavutettavuus, luettavuus ja käytettävyys. Työelämässä ohjelmistot voivat parantaa tehokkuutta ja vähentää kustannuksia. Toisaalta, jos ohjelmisto on heikko, se voi aiheuttaa lisäkustannuksia ja viiveitä. Heikko ohjelmisto voi johtaa myös eri haavoittuvuuksiin, joita kyberhyökkääjät voivat hyödyntää. Näistä voi aiheutua erittäin suuria ongelmia. Viime vuosina tutkimuksissa on todettu, että iso osa haavoittuvuuksista johtuu heikosta muistiturvallisuudesta.

Tässä opinnäytetyössä tutkitaan muistiturvallisuutta: mitä se tarkoittaa ja miten se vaikuttaa ohjelmistokehitykseen. Näihin kysymyksiin vastataan teorian ja ohjelmointiesimerkin avuin sekä syventymällä Rust-ohjelmointikieleen. Rust on muistiturvallisuuden kannalta mielenkiintoinen sen muistinhallinnan lähestymistavasta. Rust on ollut myös useamman vuoden putkeen trendaavin ohjelmointikieli sekä useat yritykset ovat kasvattaneet tehokkuuttaan vaihtamalla Rustiin.

2 Muistiturvallinen ohjelmointi

Muistiturvallisuuden tarkoitus on estää muistinhallintaan liittyviä bugeja tai virheitä, mutta ennen kuin syvennytään muistiturvallisuuteen, on hyvä tietää, mitä muistilla tarkoitetaan ohjelmoinnissa. Tietokoneen muistista puhuttaessa on tärkeää, ettei muistia sekoita tietokoneen tallennustilaan, koska muistilla ja tallennustilalla on hyvin erilaiset käyttötarkoitukset. Muisti on tietokoneen ensisijainen muisti (engl. ”primary memory”) eli RAM-muisti ja tallennustila on toissijainen muisti (engl. ”secondary memory”), joka on yleisesti joko HDD- tai SSD-levy. Muistin ja tallennustilan erona on se, että tallennustila on vakaata, joka tarkoittaa, että tallennustilassa sijaitseva data ei häviä tietokoneen sammuessa. Muisti on toisaalta epävakaata, joka menettää datansa

tietokoneen sammuessa. Muistissa säilytetään lyhytaikaista dataa, jonka avulla esimerkiksi ohjelmistot pyörivät. (1.)

Muistinhallinnan avulla kaikki tietokoneen ohjelmistot, prosessit ja itse käyttöjärjestelmä saavat tarvitsemansa osuuden muistista, jotta ne voivat toimia. Muistinhallinta pyrkii olemaan mahdollisimman tehokas, jotta tietokoneen prosessorin tehokkuus ei laskisi, mikä hidastaa prosesseja. Muistinhallinta ohjelmistokehityksessä on joko manuaalista tai automaattista, mikä riippuu ohjelmointikielestä. Joissain ohjelmointikielissä roskankerääjä etsii automaattisesti käyttämättömiä muistipalikoita ja vapauttaa ne. Toisissa kielissä taas ohjelmoija itse joutuu varaamaan tarvitsemansa muistiosuuden ja vapauttamaan sen, kun sitä ei enää käytetä (2).

2.1 Muistiturvallisuus

Muistiturvallisuuden suosio on viime vuosina kasvanut. Syynä tähän on turvallisuusongelmien yleistyminen, joiden aiheuttajana on suurimmaksi osaksi ollut muistiturvallisuuden puute (3). Microsoftin ohjelmistohaavoittuvuusraportissa vuodelta 2019 käy ilmi, että noin 70 prosenttia heidän haavoittuvuuksistansa koostuvat muistiturvallisuusongelmista. NSA suosittelee myös muistiturvallisten ohjelmointikielien käyttöä aina, kun se on mahdollista. Yhdysvaltain hallitus on tehnyt aloitteita ajaakseen ohjelmoinnin kulttuuria muistiturvallisuutta päin. (4, s. 1.)

Muistiturvallisuus on tiettyjen ohjelmointikielien ominaisuus, joka estää yleisemmät muistin käyttövirheet. Näihin käyttövirheisiin syvennytään lisää myöhemmin. Näissäkin ohjelmointikielissä on mahdollista välttyä muistin käyttövirheiltä, mutta se vaatii enemmän resursseja, ja silti suurin osa haavoittuvuuksista liittyy muistiturvallisuuteen. Muistiturvalliset ohjelmointikielet tunnistavat muistin käyttövirheet jo koodin koonnissa tai ajon aikana. Koonnissa virheet ilmoitetaan ennen ajoa. Ajon aikana ohjelma kaatuu, jos siinä huomataan muistin käyttövirheitä. (5.)

2.2 Muistiturvallisuuden tarve

Ohjelmiston huonosti toteutettu muistiturvallisuus mahdollistaa erilaisia vaaroja. Kyberhyökkääjät voivat esimerkiksi kokonaan kaataa ohjelmiston tai uudelleenohjelmoida ohjelmiston tekemään, mitä he itse haluavat. Kyberhyökkääjät eivät kuitenkaan ole ainoa haittavaikutus. Huono muistinhallinta saattaa vaikuttaa myös ohjelmiston suoriutumiskykyyn tai epäselviin kaatumisiin. (4, s. 2.)

Buffer overflow (tai out-of-bounds) on yksi tavallisimmista muistiturvallisuuden ongelmista. Tässä ohjelma laittaa liikaa dataa muistipuskuriin, jolloin ylijäämädata siirtyy seuraavaan lähimpään muistipuskuriin ja korvaa siellä olevan datan. Tämä voi aiheuttaa datan korruptoitumisen, kaataa ohjelman tai ajaa väärää koodia. Jos taulukossa on esimerkiksi viisi alkiota ja ohjelma koettaa tulostaa kuudetta alkiota, saattaa tulostuksena olla toisen taulukon ensimmäinen alkio. Tällöin ei-haluttua alkiota voidaan myös muokata, joka on suuri tietoturvariski. (5; 6.)

Use-after-free on toinen yleinen muistiturvallisuuden ongelma. Tämä ilmenee, kun ohjelma yrittää käyttää vapautetun muistin osoitinta. Osoittimen uudelleenkäyttö saattaa aiheuttaa sen aiemmin osoittaman datan korruptoitumisen tai jopa väärän koodin ajamisen. Esimerkiksi ohjelma, joka yrittää tulostaa aiemmin poistetun taulukon alkioita, saattaakin tulostaa jonkun toisen taulukon alkioita, koska se sattuu käyttämään samaa osoitinta. (5; 7.)

Muistiturvalliset ohjelmointikielet estävät kaikki muistiin liittyvät ongelmat oletusarvoisesti sekä tekevät kyberhyökkäyksistä lähes mahdottomia. Tietenkin näissäkin ohjelmointikielissä on eroja. Joissain on minimaalinen muistiturvallisuus ja toisissa on tiukempaa. Tiukan tason muistiturvallisuus saattaa hankaloittaa koodin ajoa, koska kääntäjän tulee tehdä merkittävä määrä eri muistiin liittyviä tarkastuksia. Esimerkiksi Rustissa on paljon eri tarkastuksia, jotka voivat estää ajon, mutta Rustin virheiden käsittely on hyvin käyttäjäystävällinen (9, s. 161). Kääntöpuolella, muistiturvattomissa kielissä voidaan taas ajatella, että ohjelmoija joutuu itse tekemään merkittävän määrän

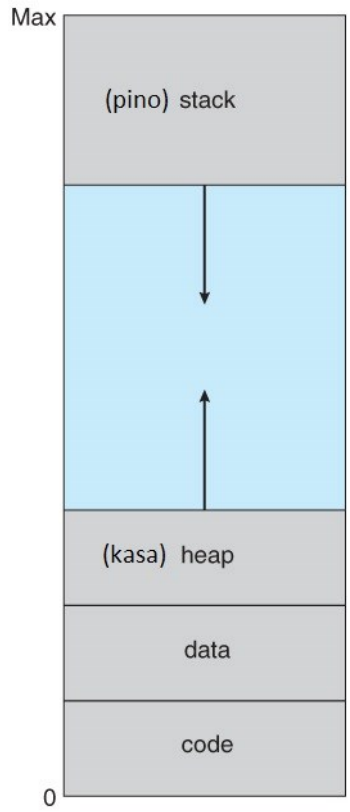
tarkastuksia, jotta ohjelma on muistiturvallinen. Yleisimmät muistiturvattomat kielet ovat C, C++ ja Assembly. (4, s. 3.)

3 Muistinhallinta ohjelmoinnissa

Jokainen ohjelmointikieli törmää muistinhallintaan. Jotkut ohjelmointikielet käyttävät roskankerääjää hoitamaan muistinhallinnan tai antavat ohjelmoijien itse huolehtia muistinhallinnasta. Ohjelmistot käyttävät ajon aikana muistia, josta ohjelmisto noutaa sille annetut ohjeet niin sanotuista eri muistin osoitteista. Muisti jaetaan kahteen osaan: pinoon (engl. ”stack”) ja kasaan (engl. ”heap”). (8.)

3.1 Pino ja kasa

Ajonaikana ohjelmisto hakee datan ja ohjeet pinosta ja kasasta. Pino ja kasa toimivat eri tavoin ja niissä säilytetään erilaista dataa. Kummallakin on omat heikkoudet ja vahvuudet, mutta niiden käyttäminen yhdessä on tehokasta. Harvoin halutaan käyttää vain toista. (8.)



Kuva 1. Muistin asettelu. (10, s. 361.)

Kuvassa 1 nähdään, miten pino ja kasa sijoittuvat muistin asetteluun. Huomataan, että alhaalla on alin tai ensimmäinen muistiosoite (nolla) ja ylhäällä suurin mahdollinen osoite (max). Pino kasvaa alaspäin ja kasa kasvaa ylöspäin. Kuvassa 1 näkyvä sininen alue on varattu pinolle ja kasalle, jota ne pystyvät täyttämään riippuen tarpeesta. Tätä aluetta täyttää pinossa ja kasassa sijaitseva data. Kasan dynaamisuuden vuoksi kasa yleensä vie suuremman muistiosuuden. (10, s. 361.)

Ohjelmoinnissa pinosegmentille varataan tietty määrä muistia. Segmentin sisällä on pinoalue, joka on dynaaminen. Ohjelman eri muuttujat ja funktiot varaavat tilansa pinoalueelta tullessaan tarpeelliseksi ohjelman suoritukseen, ja poistuu, kun tarve loppuu. Pino käyttää LIFO-menetelmää datan käsittelyyn, eli uusin data käsitellään aina ensimmäisenä. Datan haku, poisto ja lisäys on aloitettava aina pinon päällimmäisestä. Koko pino on siis käsiteltävä, jos halutaan esimerkiksi hakea pinon alin datapiste. Pinoon voidaan tallettaa vain

dataa, jonka koko tiedetään. Muuttuva tai tuntematon koko tulee tallettaa kasaan, kuten muutettavat merkkijonot. Pinon vahvuuksiin kuuluu nopeus, koska datalle ei tarvitse etsiä omaa sijaintia, vaan data laitetaan aina pinon päällimmäiseksi. Pinolla vältytään usein myös muistivuodoilta sen mallin vuoksi. Pinon ylivuotoa on kuitenkin varottava, joka ilmenee silloin, kun pinoalue täyttyy. Tämä johtaa ohjelman kaatumiseen. Ylivuoto yleensä ilmenee, kun ohjelmassa käytetään useita ja monimutkaisia tai päättymättömiä sisäkkäisiä funktioita. (8; 9, s. 59–61.)

Kasan käyttäminen on vapaampaa ja dynaamista. Kasaan datan tallettaminen varaa tietyn tilan kasamuistista ja palauttaa osoittimen, joka osoittaa takaisin tähän tilaan. Kasassa data ei ole missään järjestyksessä eikä sillä ole tiettyä muotoa, joten kasasta voidaan noutaa tai muokata dataa vapaasti. Monimutkaisemmat datan struktuurit, kuten binääripuut, sekä kaikki kokoaan muuttavat datatyypit sopivat kasaan sen dynaamisuuden vuoksi. Esimerkiksi muutettavissa olevat String-muuttujat talletetaan kasaan. Kasan heikkoutena ovat mahdolliset muistivuodot, tilan etsintä datalle sekä manuaalisen muistinhallinnan hankaluus. (8; 9, s. 59–61.)

3.2 Heikon muistinhallinnan vaarat

Ohjelmistot tarvitsevat muistia toimiakseen, eikä muisti ole loputonta ja se voi loppua kesken. Muistia tulee siis vapauttaa, jotta ohjelmisto ei törmää virheisiin tai jopa kaadu kokonaan. Muistia käyttää eri objektit, ja ne objektit, joita ohjelmisto ei enää tarvitse lopun ajonaikana (eli ns. roskat), tulisi vapauttaa, mikä antaa enemmän muistia muille tarvittaville objekteille. Ohjelmointikielet lähestyvät tätä ongelmaa joko manuaalisella tai automaattisella muistin vapauttamisella. (11, s. 1.)

Manuaalisessa vapauttamisessa on inhimillisten virheiden vaara. Suuri osa ongelmista esiintyy kasassa sijaitsevasta datasta sen dynaamisuuden vuoksi. Kasassa olevaa dataa käsitellään osoittimien avulla. Use-after-free-ongelma esiintyy, kun osoitin vapautetaan, mutta tätä osoitinta silti käytetään uudelleen. Ohjelmiston seuratessa vapautettua osoitinta ei voida tietää, mitä tapahtuu.

Yleensä päädytään ohjelmiston kaatumiseen tai virheellisiin tuloksiin, jotka voivat olla hyvin hankalasti havaittavissa. Vaarana ovat myös muistivuodot, jotka ilmenevät, kun käyttämätöntä dataa ei vapauteta. Tällöin ohjelmisto hidastuu tai jopa kaatuu, koska muisti ei riitä. Nämä vaarat ovat olennaisia esimerkiksi C-ohjelmointikielessä, jossa ohjelmoijan tulee itse käyttää free()-funktiota, joka vapauttaa muistin. (11, s. 2.)

Kilpailutilanne on yleinen ongelma ohjelmistoissa, jotka käyttävät samanaikaisuutta. Kilpailutilanteen syntyminen vaatii jaetun tilan (engl. ”shared state”), joka viittaa muuttujiin, joita säikeet pystyvät käsittelemään samanaikaisesti. Kilpailutilanne syntyy, kun useampi kuin yksi säie yrittää käsitellä samaa dataa samaan aikaan. Tässä kilpaillaan siitä, missä järjestyksessä säikeiden komennot käsitellään. Näistä tilanteista syntyy erilaisia virhetilanteita, kuten ohjelman kaatuminen, datan lukemisen ja/tai kirjoittamisen epäonnistuminen tai lukkotilanteet. Lukkotilanteessa prosessit odottavat loputtomasti toisiaan vapautumaan, mikä pysäyttää kaiken. Kilpailutilanteilta vältytään, jos jaettua tilaa ei käytetä. Toinen tapa välttyä on käyttää säikeiden synkronointia tai atomisia operaatioita. Atomiset operaatiot varmistavat, että jokainen operaatio suoritetaan ilman keskeytyksiä. Säikeiden synkronointi onnistuu lukoilla. Yksi säie lukitsee datan itselleen operaation ajaksi ja vapauttaa sen operaation lopuksi. (12.)

3.3 Roskankeräys

Roskankeräys on yksi automaattinen muistinhallintamenetelmä, jota suuri osa muistiturvallisista ohjelmointikielistä käyttää. Roskankeräys on luotu helpottaakseen ohjelmoijaa tekemästä muistinhallintaan liittyviä virheitä. Roskankeräys on pääasiallisesti aina päällä ajonaikana. Roskankeräyksen tehtävänä on etsiä muistilohkoja, joita ohjelmisto ei enää käytä ja vapauttaa ne uudelleenkäytettäviksi. Roikkuvia osoittimia ei esiinny, koska roskankeräys vapauttaa muistilohkon vasta, kun tähän muistilohkoon ei päästä enää käsiksi millään osoittimella. Muistivuodot estyvät myös roskankeräyksen ansiosta (10, s. 4–5). Roskankeräys yksinkertaistaa ohjelmointia, mikä helpottaa koodaamista, jolloin tehokkuus kasvaa. Roskankeräyksen heikkoutena on sen

vaatima osuus prosessorista, mikä hidastaa ohjelmiston toimivuutta tai jopa keskeyttää joitain ohjelmiston ominaisuuksia. Mitä isompi tai monimutkaisempi ohjelmisto, sitä enemmän roskankerääjä joutuu käymään koodia läpi varmistaakseen, mitkä muuttujat ovat roskaa. (13.)

4 Rust-ohjelmointikieli

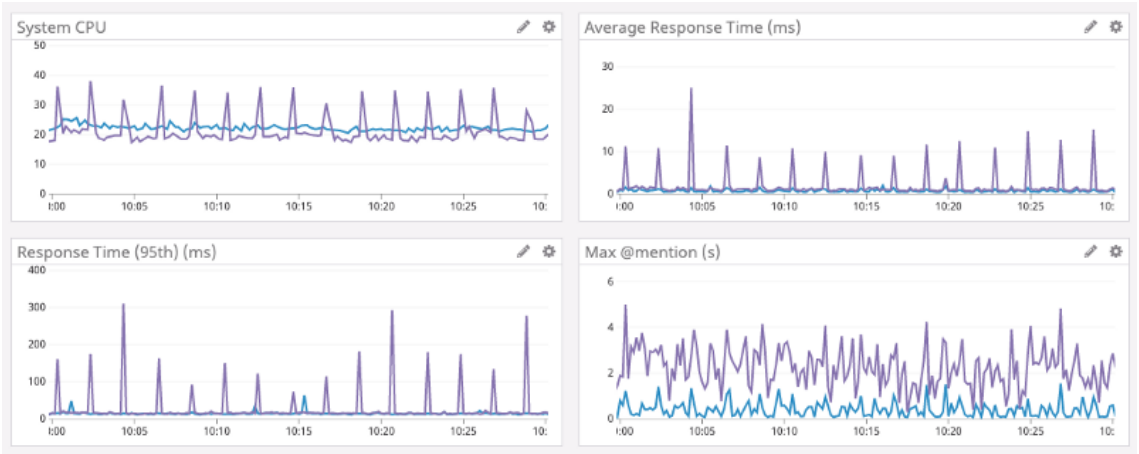
Ohjelmointikieli on tietokonekieli, jonka avulla ohjelmoija voi kirjoittaa niin sanottuja ohjeita tietokoneelle toteuttaakseen jonkun tehtävän. Yleisesti ohjelmointikieliä käytetään eri ohjelmistojen kehittämiseksi kuten työpöytä-, mobiili- tai verkkosovellukset. Ohjelmointikieliä on nykyään hyvin paljon, joten ne ovat jaettu matalan ja korkean tason ohjelmointikieliin. Kummallakin kategorialla on omat vahvuutensa ja heikkoutensa sekä alakategorioita. Pääasiallisesti matalan tason ohjelmointikielellä koodin ajaminen on nopeampaa, mutta niiden tulkinta on hankalampaa. Korkean tason ohjelmisto on helpommin tulkittavissa, joten ohjelmointi näillä kielillä on helpompaa. Korkean tason ohjelmointikielet ovatkin suositumpia kuin matalan tason kielet. (14.)

4.1 Johdanto

Rust on matalan tason ohjelmointikieli. Yleensä matalan tason ohjelmointikielet ovat hankalampia, koska niissä ohjelmoijan täytyy miettiä enemmän muistinhallintaa. Matalan tason ohjelmoijat käyttävät vuosia välttyäkseen tietyiltä muistinhallinnan vaaroilta ja silti joutuvat ohjelmoimaan varoen, jotta ohjelmistoon ei jää tietoturva-aukkoja tai muita ongelmia, kuten kaatumisia tai datan korruptoitumista. Rustin kääntäjä estää nämä vaarat, jolloin vältytään mahdollisilta ohjelmointivirheiltä. Rust pyrkii olemaan luonnollisesti tehokas muistinhallinnassa ja nopeudessa. Rustia pidetäänkin hyvänä välimaastona matalan ja korkean tason ohjelmointikielien välillä. Ei käytetä roskankeräystä, eikä ohjelmoijan tarvitse tehdä manuaalista muistinhallintaa. Vaikka Rust on matalan tason ohjelmointikieli, sitä voidaan käyttää pitkälti samoihin tarkoituksiin kuin mihin korkeatasoisia ohjelmointikieliä käytetään. (9, s. xix, xxv.)

Rust on ollut seitsemän vuotta peräkkäin (2016–2022) Stack Overflow -yhteisön suosituin ohjelmointikieli (15). Syynä tähän on se, että Rust käsittelee paremmin tiettyjä muiden ohjelmointikielten heikkouksia, mikä pitää samalla omien heikkouksien määrän siedettävänä ja lupaamalla tehokkaampaa kokonaisuutta sekä muistiturvallisuutta. Esimerkiksi Skylight-sovelluksen kehittäjät onnistuivat vähentämään sovelluksen muistikäyttöä 5 gigatavusta 50 megatavuun vaihtamalla osittain Javasta Rustiin (17). Googlen Android-käyttöjärjestelmissä on otettu Rust käyttöön, joka Googlen raporttien mukaan on vähentänyt suuresti muistihaavoittuvuuksia (3).

Suosittu viestintä- ja VoIP-sovellus Discord on vaihtanut Go-kielestä Rustiin. Discord-kehittäjät huomasivat, että Go-kielellä ilmenee käyttöpiikkejä viiveessä ja prosessorissa. Piikit johtuivat Go-kielen roskankeräyksestä, koska Discordilla on valtava määrä dataa ja roskankeräys joutuu käymään kaiken läpi, ennen kuin se voi vapauttaa muistia. Usein roskat jäävät niin sanotusti roikkumaan, koska roskankeräys ei ole vielä ehtinyt loppuun saakka, jolloin se ei tiedä, voiko muistia vapauttaa. Discord-kehittäjät tiesivät ennestään Rustin olevan erittäin muistitehokas eikä käytä roskankeräystä, joten he kokeilivat korjata piikit vaihtamalla Rustiin. (16.)



Kuva 2. Discordin käyttöpiikkivertailu Rust (sininen) vs Go (violetti) (16).

Discord-kehittäjät onnistuivat poistamaan viivepiikit ja vähentämään huomattavasti muita käyttöpiikkejä. Viivepiikit näkyvät kuvassa 2 otsikolla

”response time” ja ”average response time”. Kuvasta 2 huomataan myös, että prosessoripiikit (System CPU) ovat huomattavasti vähentyneet. Rustilla kehittäjät saavuttivat samantasoisen tehokkuuden kuin Go-kielellä vain yksinkertaisella optimoinnilla. Profiloimalla ja suorituskyvyn optimoinnilla saavutettiin kuvan 2 tulokset (16).

4.2 Rustin pääominaisuudet

Rust käyttää omanlaista lähestymistapaa muistinhallintaan, joka tekee Rustista Rustin. Tämä lähestymistapa on omistajuus ja sen eri aputoiminnot, kuten referenssit. Näiden avulla Rust pystyy lupaamaan koodin olevan aina muistiturvallista sekä pitävän tehokkuuden verrattavissa C- ja C++ -kieliin. (9, s. 59; 16.)

4.2.1 Omistajuus

Omistajuuden kautta päästään Rustin omalaatuisuuteen käsiksi. Omistajuuden tarkoitus on hallita muistia, ilman uhrauksia tehokkuudessa. Tämä saavutetaan sillä, että Rust estää ajon, jos omistajuuden sääntöjä rikotaan. Tällä estetään samalla mahdolliset virheet, jotka olisivat saattaneet tapahtua ajon aikana. Omistajuuden säännöt ovat seuraavat: jokaisella arvolla on omistaja, voi olla vain yksi omistaja kerrallaan ja omistajan poistuessa näkyvyysalueelta Rust pudottaa arvon drop()-funktiolla. Rust kutsuu tämän funktion aina sulkevan aaltosulun kohdalla, sitä ei tarvitse ohjelmoijan erikseen tehdä itse. Oletusarvoisesti drop() vapauttaa näkyvyysalueen dynaamisten muuttujien arvojen käyttämät muistiosuudet kasasta. Käydään nämä säännöt läpi yksinkertaisella esimerkkikoodilla. (9, s. 61.)

```
{
     let x = String::from(“hello world”);

     // x on alustettu ja valmis käytettäväksi

     println!("{}", x);

}
// x ei ole enää näkyvyysalueella
```

Esimerkkikoodi 1. Omistajuuden näkyvyysalue-esimerkki.

Esimekkikoodissa 1 luodaan aluksi x-muuttuja ja sille annetaan String-arvo ”hello world”. Tämän arvon omistaa muuttuja x. Kyseessä on dynaaminen String-muuttuja, joten se talletetaan kasaan. Tämän jälkeen x-muuttujaa voi käyttää, kunnes sulkeva aaltosulku tulee vastaan. Sen jälkeen omistaja ei ole enää näkyvyysalueella, jolloin arvo pudotetaan. Kun Rust huomaa, että dynaamisen muuttujan omistaja ei ole enää näkyvyysalueella, se kutsuu drop()-funktion, joka vapauttaa muuttujan käyttämän muistin. Tällä Rust estää muistivuodot sekä varmistaa aina tehokkaan kasamuistikäytön, koska muistin vapautus mahdollistaa sen uudelleen käytön, mikä estää myös vapaan muistin loppumisen (9, s. 63). C-ohjelmointikielessä ohjelmoija joutuu itse kutsumaan free()-funktion, kun haluaa vapauttaa dynaamisten muuttujien varaaman muistialueen. Staattiset eli pinomuistissa sijaitsevat muuttujat eivät tarvitse manuaalista vapautusta. Kasamuistin tehokkuus ja muistivuodot ovat C:ssä riippuvaisia ohjelmoijan tarkkuudesta (18, s. 108).

Muuttujiin pystytään viittaamaan tai muokkaamaan näkyvyysalueiden ulkopuolelta eri tavoin. Muuttujan sisältämä data voidaan siirtää sekä siirtää omistajuus, tai muuttujan datasta voidaan tehdä kopio jatkokäyttöä varten, tai lainata muuttujaa, mikä säilyttää omistajuuden. Lainaukset ovat hyödyllisiä, kun muuttujasta halutaan vain lukea arvo. Omistajuuden siirto kannattaa esimerkiksi tehdä silloin, kun jokin toinen funktio tarvitsee arvoa, mutta alkuperäinen funktio ei.

```
fn main() {
     let x = String::from(“hello world”);

     let y = x;

     println!(“{}”, y);
}
```

Esimerkkikoodi 2. Omistajuuden siirtäminen.

Esimerkkikoodissa 2 luodaan x-muuttuja ja sen arvo annetaan y-muuttujalle. Ohjelma tulostaa y:n sisällön. Yrittäessä tulostaa x-muuttujan sisältöä Rust antaa virheen, josta selviää, että omistajuus on siirtynyt ”let y = x” -kohdassa.

4.2.2 Referenssit ja lainaaminen

Referenssien avulla muuttujia voidaan lainata siirtämättä omistajuutta. Referenssin luomista kutsutaan lainaamiseksi. Referenssi luodaan laittamalla &-merkki muuttujan nimen eteen. Kuten omistajuudella, referensseilläkin on sääntöjä: vain yksi muutettava referenssi on sallittu tai muuttumattomia referenssejä saa olla yksi tai enemmän. (9, s. 71–77.)

```
fn main() {
     let x = String::from(”hello world”);

     let len = calculate_length(&x);

     println!(“Length of ‘{}’ is {}.”, x, len);
}

fn calculate_length(s: &String) -> usize {
     s.len()
}
```

Esimerkkikoodi 3. Muuttujan referointi (9, s. 71).

Esimerkkikoodissa 3 luodaan taas muuttuja x, mutta tällä kertaa syötetään x-muuttuja calculate_length()-funktioon. Tälle funktiolle kerrotaan, että sille syötetään referenssi kirjoittamalla &-merkki muuttujan eteen. Funktion parametriin pitää myös ilmoittaa, että funktiolle syötetään referenssi kirjoittamalla ”&String”. Tässä nähdään, että x-muuttujaa on käytetty, mutta omistajuus ei ole siirtynyt. Rust ei automaattisesti kutsu drop-funktiota calculate_length()-funktiossa. Tämä johtuu siitä, että omistajuus ei ole siirtynyt.

4.2.3 Lainauksien muokkaaminen

Oletusarvoisesti lainattuja muuttujia ei voida muokata, mutta muokkaaminen on mahdollista tekemällä alkuperäisestä muuttujasta muutettava. Tämä tehdään lisäämällä ”mut” eli ”mutable” muuttujan eteen muuttujan alustuksessa.

```
fn main() {
     let mut x = String::from(“hello”);

     println(“{}”, x);

     add_text(&mut x);

     println(“{}”, x);
}

fn add_text(s: &mut String) {
     s.push_str(“ world”);
}
```

Esimerkkikoodi 4. Muuttujan lainaaminen ja muokkaaminen.

Alustamalla x-muuttuja esimerkkikoodissa 4 muutettavana (mut) mahdollistetaan sen arvon muokkaaminen toisessa funktiossa. Funktioon tulee myös ilmoittaa parametrina, että sille annetaan muutettava String-muuttuja kirjoittamalla ”&mut String”. Esimerkkikoodin 4 tulosteena on ensin ”hello” ja add_text()-funktio lisää tähän ”world”, jolloin seuraava tuloste antaa ”hello world”. Huomataan, että x-muuttujaa on muokattu ilman, että sen omistajuus olisi siirtynyt. Tässäkään Rust ei kutsu drop-funktiota add_text()-funktion jälkeen, koska omistajuus ei ole siirtynyt.

4.3 Samanaikaisuus

Nykypäivänä tietokoneiden prosessoreissa on useita ytimiä, ja niiden oikeanlainen hyödyntäminen ohjelmoinnissa voi olla merkittävä tekijä tehokkuuden kasvattamiseen. Tämä onnistuu samanaikaisella ohjelmoinnilla. Samanaikaisessa ohjelmoinnissa hyödynnetään säikeitä, jotka ovat keskeisiä yksikköjä prosessorin hyödyntämisessä. Säikeet jakavat resurssinsa, koodinsa, datansa, yms. muiden saman prosessin säikeiden kanssa. Perinteisesti prosessit ovat käyttäneet yhtä säiettä mahdollistaen yhden tehtävän suorituksen

kerrallaan. Monisäikeisessä prosessissa voidaan suorittaa useampia tehtäviä kerrallaan samanaikaisesti. Ohjelmistoissa monisäikeisyys kasvattaa esimerkiksi vastauskykyä: toinen säie voi heti toteuttaa käyttäjän pyynnön, vaikka muut säikeet suorittavat muita ohjelmiston toimintoja. (10, s. 153.)

Samanaikainen ohjelmointi on tunnetusti monimutkaista ja herkkä virhetilanteille. Yleisiä ongelmia ovat lukko- ja kilpailutilanteet, jotka voivat aiheuttaa ohjelmiston kaatumisen tai odottamattomia tuloksia. Rust on mahdollistanut omistajuuden sääntöjen avulla paremman turvan yleisiä samanaikaisuuden ongelmia vastaan. Rust tunnistaa koodissa olevat samanaikaisuuden virheet koonnin aikana, eikä ajon aikana. Virhetilanteessa Rust estää ajon ja antaa virheviestin, jolloin ohjelmoijan on helppo korjata virhe. Katsotaan seuraavaksi muutama esimerkkikoodi säikeiden käyttämisestä Rustilla. (9, s. 353.)

```
use std::thread;
use std::time::Duration;

fn main() {
     let handle = thread::spawn(|| {
          for i in 1..10 {
               println!("number {} from the spawned thread!", i);
               thread::sleep(Duration::from_millis(1));
          }
     });

     for i in 1..5 {
          println!("number {} from the main thread!", i);
          thread::sleep(Duration::from_millis(1));
     }

     handle.join().unwrap();
}
```

Esimerkkikoodi 5. Säie-esimerkki (9, s. 354).

Uusi säie luodaan esimerkkikoodissa 5 thread::spawn()-funktiolla. Säie alustetaan muuttujaksi. Uusi säie tulostaa for-silmukassa numeroita. thread::sleep()-funktion avulla pakotetaan säikeen toiminta pysähtymään hetkeksi, jotta toinen säie (tässä tapauksessa pääsäie) voi toimia. Tästä syystä tulosteessa ”main thread” ja ”spawned thread” tulostavat numeronsa vuorotellen. Lopuksi kutsutaan ”handle.join().unwrap()”, jonka avulla odotetaan, että handle-säiemuuttuja lopettaa toimintansa. Tällä varmistetaan, että handle-

säiemuuttuja pystyy suorittamaan koodinsa loppuun saakka. Ilman join()-kutsua uuden säikeen toiminta keskeytyy, kun pääsäie ajaa koodinsa loppuun asti. Huomaa, että join()-funktion kutsumisajoitus voi muuttaa toimintoa: jos ”handle.join().unwrap()” kutsutaankin ennen toista for-silmukkaa, handle-säiemuuttujan for-silmukka suoritetaan kokonaan ja sitten pääsäikeen for-silmukka suoritetaan. Tällöin säikeet eivät toimikaan vuorotellen.

Resurssien jakaminen on tärkeä osa samanaikaisessa ohjelmoinnissa. Säikeet voivat keskustella toistensa kanssa siirtäen tietoa ja dataa. Tämä mahdollistaa useiden eri säikeiden toiminnan samassa muistiosoitteessa (10, s. 159). Rustilla tämä onnistuu kanavien ja jaetun tilan avulla. Kanavissa omistajuus siirtyy ja jaetussa tilassa käytetään jaettua omistajuutta. Säikeen lähettäessä arvon kanavaa pitkin omistajuus siirtyy, eikä säie voi sen jälkeen enää käyttää lähetettyä arvoa. Jaetussa tilassa useampi säie pääsee käsiksi samaan muistilohkoon, jolloin arvoja ei tarvitse erikseen lähettää toisille säikeille (9, s 367).

```
use std::sync::mpsc;
use std::thread;

fn main() {
     let (sender, receiver) = mpsc::channel();

     thread::spawn(move || {
          let val = String::from("hi");
          sender.send(val).unwrap();
     });

     let received = receiver.recv().unwrap();
     println!("Got: {}", received);
}
```

Esimerkkikoodi 6. Säikeiden välinen kommunikointi kanavan avulla (9, s. 363).

Esimerkkikoodissa 6 kanava luodaan mpsc::channel()-funktiolla, ja sille alustetaan myös lähettäjä ja vastaanottaja -muuttujat (sender, receiver). Oletusarvoisesti Rustissa kanavalla voi olla useampi lähettäjä, mutta vain yksi vastaanottaja. Seuraavaksi luodaan uusi säie, samalla tavalla kuin esimerkkikoodissa 5 paitsi sender-muuttujan omistajuus pitää siirtää uuteen säikeeseen. Tämä tapahtuu ilmoittamalla thread::spawn()-funktion parametriin ”move”. Sitten luodaan lähetettävä muuttuja, joka lähetetään send()-funktiolla.

Esimerkkikoodissa 6 unwrap()-funktio pysäyttää ajon virhetilanteessa. Lopuksi vastaanottajamuuttuja (receiver) vastaanottaa säikeestä lähetetyn muuttujan recv()-funktiolla ja tulostaa sen sisällön. Tärkeää on huomata, että sender-muuttujan arvon omistajuus siirtyy uuteen säikeeseen sekä val-muuttujan arvon omistajuus siirtyy send()-funktiokutsun jälkeen pääsäikeeseen. Koettaessa tulostaa val-muuttujan arvoa uuden säikeen sisällä lähettämisen jälkeen tuottaa virheen, koska omistajuus on siirtynyt.

Jaetussa tilassa hyödynnetään vastavuoroista poissulkua eli mutex (engl. ”mutual exclusion”). Mutex on datastruktuuri, johon voidaan tallettaa dataa. Mutexit hyödyntävät lukkoja, jotka rajoittavat säikeiden pääsyä dataan. Säikeen on ensin saatava lukko itselleen, jotta se voi päästä mutexin sisältämään dataan käsiksi. Lukko voi olla vain yhdellä säikeellä kerrallaan. Säikeen on vapautettava lukko, kun se ei enää tarvitse sen dataa. Tällöin muut säikeet pääsevät omalla vuorollaan käyttämään mutexin dataa. Rustissa jaetussa tilassa tulee myös käyttää atomista referenssiviitelaskentaa tavallisen referenssiviitelaskennan sijaan, koska tavallisessa viitelaskennassa tilan jakaminen eri säikeille ei ole turvallista. Tämä johtuu siitä, että eri säikeet voivat vahingossa keskeyttää viitelaskennan, joka voi johtaa muistivuotoihin tai aikaiseen muistin vapauttamiseen. (9, s. 367.)

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
     let counter = Arc::new(Mutex::new(0));
     let mut handles = vec![];

     for _ in 0..10 {
          let counter = Arc::clone(&counter);

          let handle = thread::spawn(move || {
               let mut num = counter.lock().unwrap();

               *num += 1;
          });
          handles.push(handle);
     }

     for handle in handles {
          handle.join().unwrap();
     }

     println!("Result: {}", *counter.lock().unwrap());
}
```

Esimerkkikoodi 7. Säikeiden käyttö jaetussa tilassa (9, s. 367).

Esimerkkikoodin 7 ohjelman alussa alustetaan counter-muuttuja nollaksi, joka on Mutex-tyyppinen sekä atominen referenssiviite eli Arc. Seuraavaksi alustetaan handles-vektorimuuttuja, joka tulee toimimaan säikeiden taulukkona. for-silmukka ajetaan 10 kertaa, jossa jokaisessa instanssissa luodaan uusi säie ja kloonataan counter-muuttuja referenssin avulla. Kloonauksella jokaisella säikeellä on oma referenssi counter-muuttujaan. Säikeen luonnissa (”thread::spawn()”) parametriin ilmoitetaan ”move”, jolla counter-muuttujan omistajuus siirtyy säikeeseen. Säikeessä luodaan muuttuva muuttuja num, joka saa mutex:n lukon lock()-funktiolla. Tämän jälkeen num-muuttujan avulla voidaan tehdä muutoksia counter-muuttujaan. Tässä tapauksessa arvoa kasvatetaan yhdellä. Huomaa, että num-muuttujan edessä on *-merkki, joka dereferoi mutex:n, jotta arvoon päästään käsiksi. Silmukan lopussa säie lisätään handles-taulukkoon. Ohjelman lopuksi toisessa for-silmukassa varmistetaan kaikkien säikeiden prosessien valmistuminen join()-funktiolla. Lopulta counter-muuttujan arvo voidaan tulostaa saamalla mutex:n lukko ja dereferoimalla. Ohjelma tulostaa arvon 10, koska ohjelmassa luodaan 10 säiettä, joista jokainen kasvattaa arvoa yhdellä.

Kanavien ja jaetun tilan avulla Rust-ohjelmoijat voivat huoletta hyödyntää samanaikaista ohjelmointia, ainakin muistiturvallisuuden näkökannalta. Rustin säännöt estävät suurimmat ongelmat, kuten kilpailu- ja lukkotilanteet, jotka muissa ohjelmointikielissä voivat esiintyä. Samanaikainen ohjelmointi kuitenkin vaatii Rust-ohjelmoijiltakin tarkkuutta ja tietotaitoa. Monisäikeinen koodi on huomattavasti monimutkaisempaa kuin yksisäikeinen.

5 Muistiturvallisen ohjelman toteutus

Tässä luvussa vertaillaan Rustin ja C-ohjelmointikielien eroja muistiturvallisuuden toteuttamiseksi. Vertailun avuksi kummallakin ohjelmointikielellä on ohjelmoitu äänilaskurisovellus, jossa demonstroidaan kummankin ohjelmointikielen käytäntöjä. Kummatkin ohjelmointikielet ovat matalan tason ohjelmointikieliä sekä niissä on huomattavia eroja muistinhallinnassa. C on yksi harvoista ohjelmointikielistä, jossa muistinhallinta on manuaalista. Rust toisaalta on poikkeama: siinä ei käytetä roskankeräystä, mutta ei myöskään manuaalista muistinhallintaa.

5.1 Äänilaskurisovelluksen kuvailu

Sovellukselle syötetään .CSV-tiedostoja, joiden koko on tuntematon. Tiedostot sisältävät äänestyksen ääniä, jotka ovat numeroita (1, 2 tai 3). Tiedostoissa on myös tunnistenumero jokaiselle äänelle. Tiedostojen asettelu on seuraavanlainen:

```
number;choice
156;3
157;2
```

Sovellus käy tiedostot vuorotellen läpi, ja kaikki tiedostossa olevat äänet tallennetaan dynaamisesti kasvavaan tietorakenteeseen. Muuttujan on oltava dynaaminen, koska sovellukselle annetaan useampi kuin yksi tiedosto sekä tiedostojen koko on tuntematon. Tällöin muuttuja tulee tallettaa kasaan. Sovellus laskee äänestyksen voittajan muuttujan avulla.

5.2 Sovelluksen toteutus Rustilla

Aluksi tiedoston käsittelyä varten alustetaan tietotyyppi CsvData. Tiedostoissa on tunnistenumero ja ääni. Tiedostojen käsittelyn avuksi käytetään Serde-kirjaston deserialisointia apuna, jonka avulla ”number”-otsikkoa ei tarvitse ilmoittaa. ”number”-otsikkoa ei kuitenkaan tarvita äänten laskemiseen. Rustille tulee kuitenkin ilmoittaa äänten (choice) tietotyyppi, joka on kokonaisluku eli i32.

```
#[derive(Deserialize)]
struct CsvData {
     choice: i32,
}
```

Esimerkkikoodi 8. Tietotyyppi CsvDatan määrittely.

Tarkastellaan seuraavaksi main()-funktiota, jossa syötetään tiedostot ja määritetään vektorimuuttuja sisältämään tiedostojen sisällöt eli äänet.

```
fn main() {
     let mut vote_storage: Vec<i32> = Vec::new();

     let file_paths =
     ["./votes1.csv", "./votes2.csv", "./votes3.csv"];

     for file_path in file_paths {
          process_files(&file_path, &mut vote_storage)?;
     }
     // …
}
```

Esimerkkikoodi 9. Tiedostojen syöttö ja vektorin määritys.

Tiedostojen sisällön säilyttämiseksi valittiin vektori, koska vektorit ovat dynaamisia. Äänten lukumäärää ei tiedetä ja tiedostoja annetaan useampi, joten koko muuttuu ajon aikana. Vektorit on allokoitava tästä syystä kasaan. Rust automaattisesti allokoi muistilohkon vektorille, jonka koko riippuu määrityksessä annetusta tietotyypistä. Tarvittaessa Rust siirtää vektorin suurempaan (tai pienempään) muistilohkoon, poistaen datan vanhasta muistilohkosta, mikä vapauttaa sen uudelleenkäytettäväksi (9, s. 142–146). Vektori alustetaan muutettavaksi (mutable eli ”mut”). for-silmukassa tiedostot ja vote_storage -vektori annetaan parametrina funktiolle process_files(), jossa tiedostojen sisällöt käydään läpi. Funktio ei ole samassa näkyvyysalueessa,

joten parametriin pitää kertoa muuttujien olevan referenssejä ja vote_storage-muuttujan lisäksi olevan muutettava (&mut). &-merkillä kerrotaan kääntäjälle, että muuttujaa lainataan, jolloin omistajuus ei siirry. process_files()-funktio on seuraavanlainen:

```
fn process_files(file_path: &str, storage: &mut Vec<i32>)
-> Result<(), io::Error> {
     let file = File::open(file_path)?;
     let mut reader =
     csv::ReaderBuilder::new().delimiter(b';').from_reader(file);

     for record in reader.deserialize() {
          let record: CsvData = record?;
          storage.push(record.choice);
     }

     Ok(())
}
```

Esimerkkikoodi 10. process_files()-funktio.

Esimerkkikoodissa 10 olevan process_files()-funktion tarkoituksena on avata parametrina saatu tiedosto, käydä se läpi ja syöttää sisältö parametrina saatuun vektorimuuttujaan. Tiedostoja sovellukselle syötetään kolme kappaletta, joten process_files()-funktio kutsutaan kolmesti. Funktio kutsutaan for-silmukassa, joten tiedostot käsitellään vuorotellen, ei samanaikaisesti. Parametreina on kaksi referenssiä, toinen String-muuttujalle eli tiedostolle ja toinen muutettavalle vektorille. Lukijan ja Serde-kirjaston deserialize()-funktion avulla tiedosto käydään rivi riviltä läpi for-silmukassa. ”let record: CsvData = record?;’ -komento muuntaa tiedoston joka rivin CsvData-tietotyyppiin, jotta tiedoston sisältöä voidaan käsitellä. Kysymysmerkillä varmistetaan, että tiedoston läpikäynti estetään virhetilanteessa ja viimeistään tiedoston rivien loputtua. Sitten joka rivin sisältö sijoitetaan vektoriin. Vektorin koko kasvaa dynaamisesti sitä mukaa, kun sille syötetään tiedostojen sisältöjä. Lopulta ilmoitetaan kaiken menneen hyvin Ok(()):lla.

Seuraavaksi vote_storage-muuttuja syötetään process_votes()-funktiolle parametrina. Tämä funktio laskee vektorin äänet ja tulostaa voittajan.

```
fn main() {
     // …
     process_votes(&vote_storage);
}

fn process_votes(storage: &Vec<i32>) {
     let total_votes = storage.len();
     let mut votes = [0; 3];

     for vote in &storage[..total_votes] {
          match vote {
               1 => votes[0] += 1,
               2 => votes[1] += 1,
               3 => votes[2] += 1,
               _ => println!("Invalid vote."),
          }
     }
     println!("Votes: {:?}", votes);

     let winner = votes[0].max(votes[1]).max(votes[2]);
     println!(
          "Total votes: {} and the Winner had {} votes",
          total_votes, winner);

     match winner {
          _ if winner == votes[0]
               => println!("Candidate 1 is the winner"),
          _ if winner == votes[1]
               => println!("Candidate 2 is the winner"),
          _ => println!("Candidate 3 is the winner"),
     }
}
```

Esimerkkikoodi 11. process_votes()-funktio.

Tähän funktioon riittää parametrina vain referenssi vektoriin. Muutettavaa referenssiä ei tarvita, koska tässä funktiossa siihen ei tehdä muutoksia. Äänille alustetaan uusi taulukko ”votes”, jossa on kolme elementtiä. Funktio käy parametrina saadun vote_storage-vektorin for-silmukassa läpi. Silmukassa kasvatetaan votes-taulukon elementtejä sitä mukaa, kun vektorista saatu ääni osuu kohdalle. Voittaja lasketaan vertaamalla votes-taulukon elementit max()-funktiolla. Voittaja tulostetaan match-lausekkeen avulla.

5.3 Sovelluksen toteutus C:llä

Seuraavaksi äänilaskurisovellus tehdään C:llä. Tarkoituksena on demonstroida C-kielen muistinhallintakäytäntöjä kuten malloc() ja free().

Sovellukselle syötetään aluksi tiedostot. Seuraava vaihe on tallettaa tiedostojen sisällöt muuttujaan. Aloitetaan katselemalla muuttujan, CsvData-tietotyypin ja tiedostojen alustamiset.

```
typedef struct {
     int number;
     int choice;
} CsvData;

int main() {
     FILE *file1 = openCsvFile(“votes1.csv”);
     FILE *file2 = openCsvFile(“votes2.csv”);

     int size = 10;
     int records = 0;
     CsvData *votes = (CsvData *)malloc(size * sizeof(CsvData));

     if (votes == NULL) {
          printf("Memory allocation failed.\n");
          return 1;
     }
     // …
}
```

Esimerkkikoodi 12. CsvData-tietotyypin, muuttujien ja tiedostojen alustamiset.

CsvData-tietotyyppiin tulee alustaa kaikki tiedoston rivit, ja ne alustetaan kokonaisluvuiksi. Tämä tehdään kertomaan C:lle, että luettavissa tiedostoissa on tietyn tyyppistä dataa ja niiden käsittelemiseen tarvitaan oman struktuuri. Tiedostot syötetään openCsvFile()-funktiolle, jossa varmistetaan niiden kunnollinen avaaminen sekä poistetaan tiedostossa olevat otsikot.

Seuraavaksi esimerkkikoodin 12 main()-funktiossa alustetaan size-muuttuja, ja sille annetaan arvoksi 10. Tällä muuttujalla kerrotaan malloc()-funktiolle, kuinka monta CsvData-tyyppiä syötetään. Arvo on osittain satunnaisesti valittu arvo, mutta sen tarkoitus on olla tarpeeksi iso, jotta muistilohkon kasvattamista ei tarvitsisi tehdä liian usein. Arvon ei kannattaisi olla myöskään liian iso, jolloin muistilohko saattaisi olla turhan suuri. Äänten lukumäärä tallennetaan records-muuttujaan. Seuraavaksi alustetaan osoitinmuuttuja ”*votes”, joka on CsvData-tyyppinen. Tänne tallennetaan itse äänet. votes-muuttujan alustamisessa kutsutaan malloc()-funktio (eli ”memory allocation”). Tällä kerrotaan, että tarvitaan muistia. Parametreiksi malloc() haluaa määrän (size) kerrottuna

tyyppimuoto, jotta se tietää, minkä kokoinen muistilohko alustetaan (18, s. 102). Tämän jälkeen malloc() palauttaa osoitteen muistilohkoon. Osoitinmuuttuja ”votes”:n kautta päästään muistilohkon alkuun, jossa tiedostojen äänet sijaitsevat. votes-muuttuja toimii käytännössä samalla tavalla kuin taulukkomuuttuja (array). Mahdollisen muistin allokointivirheen vuoksi if-lause sulkee sovelluksen.

Seuraavaksi sovellus käsittelee tiedostojen sisällöt processVotes()-funktiossa.

```
int processVotes(
     FILE *file, CsvData **votes, int *size, int *records) {

     while (fscanf(file, "%d;%d", &(*votes)[*records].number,
               &(*votes)[*records].choice) == 2) {

          (*records)++;

          if (*records == *size) {
               *size *= 2;

               CsvData *temp = realloc(
                    *votes, (*size) * sizeof(CsvData));
               printf("Memory reallocation successful.\n");
               if (temp == NULL) {
                    printf("Memory reallocation failed.\n");
                    fclose(file);
                    return 0;
               }
               *votes = temp;
          }
     }
     fclose(file);
     return 1;
}

int main() {
     // …
     processVotes(file1, &votes, &size, &records);
     processVotes(file2, &votes, &size, &records);
     // …
}
```

Esimerkkikoodi 13. processVotes()-funktio, jossa tiedostojen sisällöt käydään läpi.

Esimerkkikoodissa 13 processVotes()-funktio käy tiedostojen sisällöt läpi while-silmukalla. Parametreina funktiolle annetaan tiedosto, osoitinmuuttuja, koko ja äänien määrä. Silmukka pyörii niin pitkään, kun annetussa tiedostossa on rivejä. Jokaisen rivin kohdalla talletetaan valinta eli itse ääni votes-muuttujaan.

Äänet menevät omaan lohkoon taulukossa records-muuttujan avulla ja sen avulla tarkastetaan myös, tarvitaanko lisää tilaa. Silmukan pyöriessä records kasvaa, ja aina kun records on yhtä suuri kuin size, niin size tuplaantuu ja pyydetään realloc()-funktiolla lisää muistia. Realloc() (eli ”reallocation”) on funktio, jolla voidaan muuttaa aiemmin allokoidun muistilohkon kokoa. Parametreihin realloc() tarvitsee muokattavan muistilohkon osoittimen, määrän ja tyyppimuodon. Käytettäessä realloc()-funktiota on tärkeää luoda väliaikainen osoitinmuuttuja (”*temp”), jotta alkuperäinen osoitinmuuttuja ei virhetilanteessa korruptoidu eikä muistivuotoja synny (18, s. 105). Jos kaikki menee virheettömästi, niin votes-muuttujalle annetaan väliaikaisen osoitinmuuttujan osoite. Tiedosto tulee sulkea, kun sitä ei enää tarvita. Tässä tapauksessa se suljetaan while-silmukan jälkeen. Tulosteen avulla selviää, että realloc() kutsutaan ohjelman aikana viidesti.

Nyt tiedostot on käyty läpi, ja enää tarvitsee laskea lopputulos. votes ja records -muuttujat annetaan parametreina calculateVotes()-funktiolle, joka käsittelee ja laskee äänet. Funktio on seuraavanlainen:

```
void calculateVotes(CsvData *array, int records){
     int voteCounts[3] = {0};

     for (int i = 0; i < records; i++){
        int choice = array[i].choice;
        if (choice >= 1 && choice <= 3){
            voteCounts[choice - 1]++;
        }
     }
     printf("Total votes: %d\n", records);
     for (int i = 0; i < 3; i++){
        printf("Votes for %d: %d\n", i + 1, voteCounts[i]);
     }

     int winner =
     INDEX_OF_MAX3(voteCounts[0], voteCounts[1], voteCounts[2]);

     printf(“Candidate %d is the winner”, winner);
}

int main() {
     // …
     calculateVotes(votes, records);
     free(votes);
     votes = NULL;

     return 0;
}
```

Esimerkkikoodi 14. Äänten laskenta calculateVotes()-funktiolla.

Esimerkkikoodin 14 calculateVotes()-funktiolle syötetään parametrina osoitin, jonka avulla funktio saa äänet laskettavaksi. Äänet sijoitetaan voteCounts-taulukkoon. for-silmukan avulla käydään osoittimesta hakemassa äänet ja ne tarkastetaan vielä oikeiksi ennen taulukkoon syöttämistä. Komennolla ”voteCounts[choice - 1]++;” varmistetaan oikeanlainen taulukon indeksointi. Voittaja päätetään makrolla (”INDEX_OF_MAX3”), jolla vertaillaan parametrina annettujen muuttujien arvot. Tämän avulla tulostetaan voittaja. Lopuksi votes-muuttuja vapautetaan main-funktiossa käyttäen free()-funktiota.

free()-funktio vapauttaa parametrina saadun osoittimen muistilohkon uudelleenkäytettäväksi seuraaville malloc()-kutsuille. Kun free() kutsutaan, pitää muistaa, että sille annettua osoitinta ei voi enää käyttää. Hyvä käytäntö on ”nollata” osoitinmuuttuja NULL-arvolla vapauttamisen jälkeen, jotta vältytään use-after-free-tyyppisiltä virheiltä. (18, s. 108.)

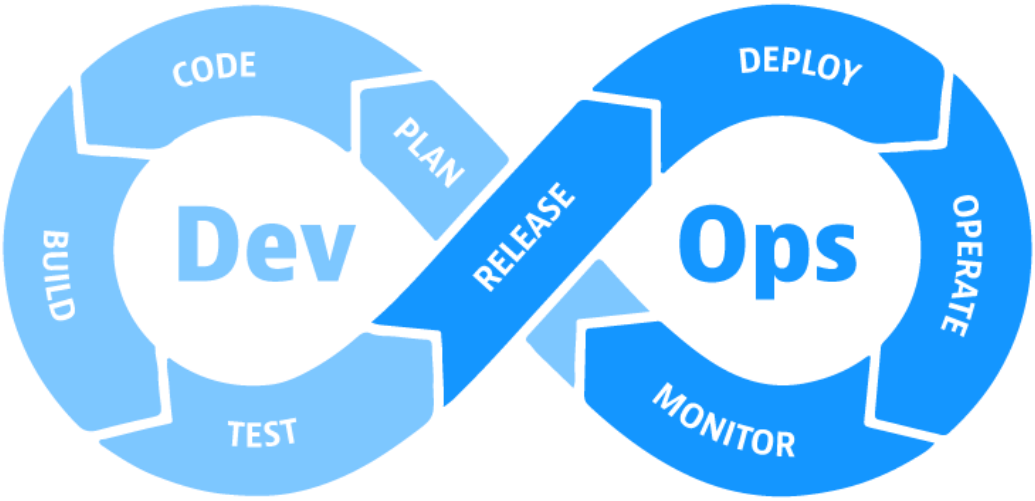
5.4 Pohdinta

Muistiturvallisuuden nimissä huomattavin ero on C-kielen mahdolliset inhimilliset virheet manuaalisen muistinhallinnan takia. Ohjelmoijan pitää varoa, ettei esimerkiksi unohda muistin vapauttamista, osoittimien käyttöä vapautuksen jälkeen eikä varaa liian pientä tai suurta muistilohkoa. Näiden tapahtuessa voi ilmentyä sovelluksen hidastumista, kaatumisia tai vääriä tuloksia. Jos realloc()-kutsua ei tehdä esimerkkikoodissa 13, niin ei saada enää minkäänlaista tulosta, ei tule edes virheviestiä. Tämä johtuu sitä, että esimerkkikoodissa 12 malloc()-funktiolla ei alusteta tarpeeksi muistia tuleville tiedostoille. Harvoin tiedetään etukäteen, paljonko muistia tarvitaan, siksi realloc() onkin tarpeellinen. C:ssä täytyy myös olla tarkkana taulukoiden indeksoinnissa. C ei ajonaikana tarkasta mahdollisia virheellisiä taulukon indeksikutsuja. Ks. esimerkkikoodi 14, jos tässä for-silmukassa käytettäisiinkin ”i <= …”, ohjelma ei tuota virhettä vaan jatkaa ajoa normaalisti. Rustissa tulee niin sanottu panikointi, joka pysäyttää ajon ja antaa virheen. C:ssä ohjelma jatkuu, mutta ei voida olla enää varmoja, mitä tapahtuu. Kasamuistista saattaa löytyä jokin toinen osoite, joka vastaa väärää indeksikutsua. Tämä saattaa aiheuttaa datan korruptoitumista, ohjelman kaatumista tai vääriä tuloksia.

Rustilla toisaalta ohjelmoijan ei tarvitse ajatella näitä edellä mainittuja asioita. Katso esimerkkikoodia 10, jossa vektoriin syötetään tiedoston sisältö, ja vektori kasvaa sitä myötä. Ohjelmoijan ei tarvitse itse kasvattaa muistilohkoa eikä vapauttaa sitä, vaan Rust tekee sen itse. Voidaan olettaa, että muistin käyttö on aina optimaalista. Tosin Rustin käytännöt ja säännöt korottavat kielen oppimiskäyrää. Omistajuuden ja referenssien käyttö ei tule välttämättä luonnostaan. Rustin kääntäjä on myös ankara, estäen ajon huomatessaan virheitä – tällä tosin estetään samalla inhimilliset virheet. Rustin virhetulosteet ovat välillä hankalia ymmärtää, esimerkiksi taulukon väärää tai mahdotonta alkiota kutsuessa ei välttämättä käy selväksi, missä tämä kutsu on tehty. Jotkin virheet Rust osaa paikallistaa paremmin, kuten &-merkin puuttuminen lainaustilanteessa tai muutettavuuden (”mut”) puute muuttujasta. Inhimillisiä muistiturvallisuusvirheitä on kuitenkin huomattavasti hankalampi tehdä Rustilla ohjelmoidessa kuin C:llä.

6 DevOps ja muistiturvallisuus

DevOps on jatkuvan tuotannon käytäntö työelämässä, joka yhdistää kehitys- ja operaatiotiimit. Lyhenne tuleekin englannin kielen sanoista ”development” ja ”operations”. DevOpsin tarkoituksena on kasvattaa näiden tiimien yhteistyötä ja parantaa työnkulkua. DevOpsin avulla yhtiöt pystyvät tuottamaan laadukkaampia sovelluksia nopeammin vähemmällä vaivalla. Keskipisteenä DevOpsissa on jatkuvuus ja automaatio. Työnkulku usein visualisoidaan jatkuvuussilmukalla. (19.)



Kuva 3. DevOps-jatkuvuussilmukka (19).

Silmukan dev-puolella tapahtuu varsinainen sovelluksen suunnittelu, toteutus ja testaaminen. Ops hoitaa hallinnollisen puolen, ylläpitää tarvittavan infrastruktuurin kehitykselle sekä monitoroi toimintaa. Tyypillisesti DevOps-projekteissa edetään pienissä osissa, jolloin silmukka käydään useasti projektin aikana lävitse. Tällä varmistetaan projektin laatu, vältytään isommilta bugeilta, saavutetaan lopputulos, joka vastaa vaatimuksia sekä mahdollistetaan vaatimusten muuttuminen kesken projektin. DevOps on hyvin ketterä

kehitystyylinen, ja saakin alkuperänsä Agile-kehitysmenetelmästä. DevOpsia yleensä hyödynnetäänkin jonkun ketterän kehitysmenetelmän kanssa. (19.)

6.1 Tietoturva DevOpsissa

Normaalisti DevOpsissa tehty projekti ei sisällä tietoturvatiimiä. Tietoturva astuu kuvioon vasta projektin loppuosassa ja lisää projektiin tietoturvaa. Nykypäivänä on paljon jatkuvaa kehitystä ja kehityssyklit ovat usein viikkoja, eikä kuukausia tai vuosia. Tietoturvaa tarvitaan siis useammin, joka normaalissa DevOpsissa aiheuttaa viivästyksiä. Tästä syystä tietoturva on lisätty mukaan DevOpsin silmukkaan. Tälle on annettu nimi DevSecOps. ”Sec” tulee englannin kielen sanasta ”security”, jolla tarkoitetaan kyberturvallisuutta tai tietoturvaa. Lisäämällä tietoturva DevOpsiin tietoturvaongelmat voidaan huomata jo kehityksen aikana, eikä vasta sen jälkeen. DevSecOpsissa tietoturvaa testataan jatkuvasti ja haavoittuvuuksien ilmaantuessa ne korjataan välittömästi. Toinen variaatio on SecDevOps, jossa tietoturva on korkeimpana prioriteettina, mutta kummankin tavoite on sama: tietoturvallinen sovellus. Ohjelmoijille opetetaan ohjelmointikäytäntöjä, joilla vältytään kokonaan haavoittuvuuksilta, esimerkiksi SQL-injektiohyökkäyksiltä tai toisessa luvussa mainituilta muistihaavoittuvuuksilta. (21; 22.)

6.2 DevSecOps menetelmät

DevSecOpsissa tietoturva otetaan heti kehityksen alussa mukaan. Suunnitellessa sovellusta voidaan jo ottaa huomioon mahdolliset riskit uhkamallinnuksella sekä ohjelmointikielen valinnassa. Uhkamallinnuksella tunnistetaan etukäteen mahdolliset haavoittuvuudet, ja sitä kautta ne voidaan estää jo ennen kuvan 3 silmukan build-vaihetta. Muistiturvallisella ohjelmointikielellä usein vältytään muistiin liittyviltä haavoittuvuuksilta sekä ohjelmoijat säästyvät muistinhallinnan vaatimalta tarkkuudelta ja ajan tarpeelta. Muistinhallintaan tulee aina panostaa, vaikka käytössä oleva ohjelmointikieli olisikin muistiturvallinen. Muistiturvattomissa kielissä muistinhallintaan tulisi panostaa erityistä enemmän. Muistinhallintaan liittyvät viat voivat usein olla

hankalia tunnistaa, joka kuluttaa resursseja. Nämä viat voivat olla myös mahdollisuus kyberhyökkääjille tekemään tuhoja. (23.)

Ohjelmoinnissa voidaan hyödyntää eri tapoja tietoturvallisen sovelluksen kehittämiseen. Näitä ovat esimerkiksi eri kehitysympäristöjen tarjoamat tietoturvalisäosat, jotka ilmoittavat heikoista tietoturvallisista koodipätkistä. Testaamalla automaattisesti koodia sekä versionhallintaa, ja muita mahdollisia käytössä olevia lisäteknologioita kuten kontteja, infrastruktuuria, pilvipalvelua, voidaan taas tunnistaa lisää haavoittuvuuksia ja korjata ne. Itse koodin pitäisi olla kirjoitettu turvallisuus mielessä, joka tarkoittaa erityistä tarkkuutta muistinhallinnassa ja olla luottamatta tavoitekäyttäjään. Esimerkiksi välttämällä vapaiden syöttökenttien käyttöä, jolloin käyttäjä ei pysty edes yrittämään SQL-injektiohyökkäystä (22).

Lopulta, kun sovellus päätyy tuotantoon, sille voidaan harjoittaa valkohattuhakkerointia. Tällöin sovelluksesta koitetaan tarkoituksellisesti löytää tietoturva-aukkoja tai heikkouksia, joita oikeat kyberhyökkääjät voisivat hyödyntää. Tätä kutsutaan myös penetraatiotestaamiseksi. Tämä on erittäin hyvä tapa vahvistaa sovelluksen tietoturvaa. (23.)

7 Yhteenveto

Työn tarkoituksena oli tutkia mitä muistiturvallisuudella tarkoitetaan, mitä hyötyä siitä on varsinkin ohjelmoinnissa ja miten Rust liittyy asiaan. Lisäksi tutkittiin Rustin toimintoja ja mitä Rust tekee eri tavalla muihin ohjelmointikieliin nähden. Myös DevOpsiin perehdyttiin, ja syvennyttiin miten DevOpsissa huomioidaan tietoturva. Näihin kysymyksiin vastattiin katsastamalla, miten muisti toimii ohjelmoinnissa, mitä haittoja ilmenee, jos muistinhallinta on heikkoa ja tehtiin äänilaskurisovellus Rustilla ja C:llä. Äänilaskurisovelluksesta kävi ilmi Rustin ja C:n erot muistinhallinnassa ja mitä kummallakin ohjelmointikielellä tulee tehdä, että sovellus on muistiturvallinen.

Päättäessä sovellusta muistiturvallisuuden demonstroimiseksi oli hankala etukäteen nähdä, miten muistiturvallisuus näkyy muistiturvallisessa

ohjelmointikielessä kuten Rustissa. Toisaalta C-kielessä selvästi näkyy, mitä muistiturvattomassa kielessä tulee tehdä muistinhallinnan kannalta, mikä antaa hyvän kontrastin muistiturvalliseen kieleen.

Opinnäytetyö oli erittäin opettavainen prosessi. Prosessin aikana perehdyin moneen eri ohjelmointialan aiheisiin kuten muistiin, tietoturvaan, ohjelmointiprosesseihin. Opettelin myös kaksi uutta ohjelmointikieltä: Rustin ja C:n, joista Rustiin perehdyin syvemmin.

Lähteet

1 Sheldon, Robert. 2022. Definition: memory management. Verkkoaineisto. TechTarget. <https://www.techtarget.com/whatis/definition/memory-management>. Päivitetty 06/2022. Luettu 11.12.2023.

2 Gillis, Alexander. Definition: memory. Verkkoaineisto. TechTarget. <https://www.techtarget.com/whatis/definition/memory>. Päivitetty 10/2020. Luettu 11.12.2023.

3 Caballar, Rina D. 2023. The Move to Memory-Safe Programming. Verkkoaineisto. IEEE Spectrum. <https://spectrum.ieee.org/memory-safe-programming-languages>. 20.03.2023. Luettu 01.11.2023.

4 Software Memory Safety. 2022. Verkkoaineisto. National Security Agency NSA. <https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF>. 11/2022. Luettu 06.11.2023.

5 Gaynor, Alex. 2019. Introduction to Memory Unsafety for VPs of Engineering. Verkkoaineisto. <https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/>. 12.08.2019. Luettu 04.11.2023.

6 Buffer Overflow. 2019. Verkkoaineisto. OWASP. <https://owasp.org/www-community/vulnerabilities/Buffer_Overflow>. Luettu 07.11.2023.

7 Using freed memory. 2019. Verkkoaineisto. OWASP. <https://owasp.org/www-community/vulnerabilities/Using_freed_memory>. Luettu 07.11.2023.

8 Nanos, Georgios. 2023. What and Where Are the Memory Stack and Heap? Verkkoaineisto. Baeldung. <https://www.baeldung.com/cs/memory-stack-vs-heap>. 11.05.2023. Luettu 24.02.2024.

9 Klabnik, S. & Nichols, C. 2022. The Rust Programming Language, 2nd edition. No Starch Press.

10 Silberschatz, A. & Galvin, P. B. & Gagne, G. 2008. Operating System Concepts, 8th Edition. Wiley.

11 Jones, R., Hosking, A. & Moss, E. 2023. The Garbage Collection Handbook, 2nd Edition. Chapman and Hall/CRC.

12 Baeldung. 2024. What is a Race Condition? Verkkoaineisto. Baeldung. <https://www.baeldung.com/cs/race-conditions>. 18.03.2024. Luettu 02.04.2024.

13 Sheldon, Robert. Garbage collection (GC). Verkkoaineisto. TechTarget. <https://www.techtarget.com/searchstorage/definition/garbage-collection>. Luettu 20.02.2024.

14 Tuama, Daragh. What Is A Programming Language? Verkkoaineisto. CodeInstitute. <https://codeinstitute.net/global/blog/what-is-a-programming-language/>. Luettu 23.01.2024.

15 Sible, J., & Svoboda, D. (2022). Rust Software Security: A Current State Assessment. Verkkoaineisto. Software Engineering Institute. <https://doi.org/10.58012/0px4-9n81>. 12.12.2022. Luettu 5.02.2024.

16 Howarth, Jesse. 2020. Why Discord is switching from Go to Rust. Verkkoaineisto. Discord. <https://discord.com/blog/why-discord-is-switching-from-go-to-rust>. 04.02.2020. Luettu 25.03.2024.

17 Goulding, Jake. 2020. What is Rust and why is it so popular? Verkkoaineisto. StackOverflow. <https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/>. 20.01.2020. Luettu 23.01.2024.

18 Seacord, Robert. 2020. Effective C. No Starch Press.

19 Bigelow, S. J. & Courtemanche M. & Gillis A. S. 2024. What is DevOps? The ultimate guide. Verkkoaineisto. TechTarget. <https://www.techtarget.com/searchitoperations/definition/DevOps>. 02/2024. Luettu 13.03.2024.

20 Gunja, Saif. 2023. What is DevOps? Verkkoaineisto. Dynatrace. <https://www.dynatrace.com/news/blog/what-is-devops/>. 16.02.2023. Luettu 13.03.2024.

21 What is DevSecOps? Verkkoaineisto. IBM. <https://www.ibm.com/topics/devsecops>. Luettu 23.03.2024.

22 Nidecki, Tomasz A. 2022. DevSecOps vs. SecDevOps. Verkkoaineisto. Acunetix. <https://www.acunetix.com/blog/web-security-zone/devsecops-vs-secdevops/>. 28.02.2022. Luettu 23.03.2024.

23 What is DevSecOps. Verkkoaineisto. Microsoft. <https://www.microsoft.com/en-us/security/business/security-101/what-is-devsecops >. Luettu 25.03.2024.