Rust för inbyggda system

Erik Ehrström 1901601
Kandidatavhandling i datateknik
Handledare: Jerker Björkqvist
Fakulteten för naturvetenskaper och teknik
Åbo Akademi
28.03.2023

Referat

Inbyggda system har blivit en del av vardagen och förkommer i allt från hemelektronik till styrenheter i flygplan. Med allt högre ansvar och flera tillämpningar står systemen inför flera säkerhetsrisker som ofta beror på programvaran. Det är viktigt att säkerställa att program, speciellt i säkerhetskritiska system, fungerar på ett säkert och effektivt sätt. Problem som buffertöverfyllningar och avreferering av nullpekare är exempel på fel som uppstår i inbyggda system på grund av fel i programmen.

Det mest förekommande programmeringspråket vid utveckling av inbyggda system är språket C, som är känt för att vara både effektivt och mångsidigt, men samtidigt är språket svårt att skriva och C garanterar inte heller minnessäkerhet. Rust är ett är modernt programmeringsspråk som garanterar minnes- och typsäkerhet utan att kräva en sophanterare för frigörande av minne. Rust kan användas inom många tillämpningsområden, men i inbyggda system begränsas användningen av brist på verktyg och stöd för olika arkitekturer. Rust genererar dessutom större exekverbara filer, vilket kan vara ett problem i resursfattiga system.

Rust ser ut att ha en framtid inom säkerhetskritiska system och möjligtvis inom inbyggda system. För att klara av att ersätta C och C++ krävs utveckling och optimering. Dessutom måste Rust växa och bli mera använt för professionella ändamål för att förhindra andra moderna programmeringsspråk att bli det nya populär språket för inbyggda system.

Sökord

Inbyggda system, minnessäkerhet, programvaruutveckling, Rust

Innehållsförteckning

1 Introduktion **1**

2 Inbyggda system **2**

2.1 Utmaningar vid utveckling . . . 3

3 Introduktion till Rust **3**

3.1 Variabler, datatyper och funktioner . . . 4

3.2 Ägarskapsmodellen . . . 7

3.3 Föränderlighet . . . 9

3.4 Felhantering . . . 9

3.5 Parallellism . . . 9

3.6 Jämförelse mellan Rust och andra språk . . . 10

3.6.1 Programmeringsparadigmer . . . 10

3.6.2 Produktivitet och verktyg . . . 11

3.6.3 Hastighet och effektivitet . . . 13

4 Rust i inbyggda system **15**

4.1 Fördelar . . . 15

4.1.1 Säkerhet . . . 15

4.1.2 Användarvänlighet . . . 16

4.1.3 Prestanda . . . 17

4.2 Utmaningar och begränsningar . . . 17

4.2.1 Begränsade plattformer och verktyg . . . 17

4.2.2 Storleken på programmen . . . 18

4.3 Förbättringar . . . 19

5 Diskussion **20**

6 Referenser **22**

1 Introduktion

Tillämpningar av mjukvara i olika former förekommer i hela vårt samhälle. En stor del av produkter som används i vår vardag har fått nya egenskaper i och med att datorer integrerats i produkterna. Datorerna är väldigt små och då de placeras in i produkter utformas ett så kallat inbyggt system. Inbyggda system har också blivit vanligare i kritiska tillämpningar, exempelvis i defibrillatorer och annan medicinsk utrustning. I de kritiska tillämpningarna är det speciellt viktigt att programmen i systemen är pålitliga, effektiva och säkra [1].

I takt med att ny elektronik har utvecklats har också nya programmeringsspråk utformats. De moderna programmeringsspråken har ofta egenskaper som typsäkerhet, vilket betyder att variabler och annan data är av det förväntade typen. Minnessäkerhet innebär att koden är gjord för att garantera att ett program inte använder minne som inte är en del av programmet vid ett visst tillfälle. Nya programmeringsspråk kan med olika tekniker uppnå minnessäkerhet. En hög minnessäkerhet kan dock påverka prestandan negativt. Ett stort problem med inbyggda system är att de sällan kan garantera minnessäkerhet. Det här medför risker som oväntade beteenden och buggar, som kan påverka systemet under drifttid. Microsoft medger att upp till 70% av deras säkerhetsrisker varje år beror på problem i minneshanteringen [2]. Nyttan av minnessäkra program är därför stor, men i och med att denna egenskap gör programmen långsammare kan det vara dyrare att använda.

Rust är ett modernt programmeringsspråk som garanterar minnes- och typsäkerhet utan att påverka hastigheten. Rust har blivit populärt bland utvecklare de senaste åren på grund av dess effektiva användning av resurser och säkra hantering av minnet. Rust används inom många områden, varav ett är programmering av inbyggda enheter. Efter år 2018 har inbyggda system varit ett av de centrala utvecklingsområdena för Rustteamet [2][3].

Denna avhandling presenterar risker och utmaningar för inbyggda system. Rust presenteras som ett alternativ för säker och effektiv utveckling av inbyggda system. Förutom att Rusts syntax, egenskaper och tillämpningar presenteras, jämförs Rust med andra programmeringsspråk för att se vilka fördelar och nackdelar som språket för med sig vid utveckling av inbyggda system med olika krav och arkitekturer.

2 Inbyggda system

Inbyggda system är datorer som är gjorda för specifika ändamål och är integrerade i större system där de utför olika funktioner. Systemen består ofta av komponenter för att kontrollera data som flödar in och ut ur systemet. Inbyggda datorer har också minne och andra hårdvarukomponenter. Komponenterna bildar ofta en mikrokontroller eller annan likande dator, där resurserna är begränsade. En mikrokontroller är en liten dator på ett kretskort med olika komponenter och en processor. Eftersom flera typer av datorer kan användas för inbyggda system är definitionen inte entydig. Vissa definitioner klassificerar mobiltelefoner som inbyggda system, medan andra har kriterier som att de måste utföra specifika uppgifter för andra system eller att de inte får använda konventionella operativsystem som Linux eller Windows. [4]. Den här avhandlingen tar inte ställning till definitionen för inbyggda system, men mjukvaran som diskuteras är gjord för system med begränsade resurser.

Inbyggda system används i många olika former som i konsumentelektronik, industriella kontrollsystem, automatisering och kontrollenheter i fordon, för medicinsk utrustning och för militärt bruk. Systemen förekommer allt oftare i både kritiska system och system som är mindre kritiska. Kristiska tillämpningar kan exempelvis vara automatiserade kontollsystem i flygplan. Mindre kritiska tillämpningar finns ofta i vardagliga produkter, t.ex. i hemelektronik för att ge produkter flera egenskaper [4][5]. I flera av tillämpningar av inbyggda system har enheterna utvecklats och blivit mer komplexa och krävande. I takt med att systemen blivit mer avancerade har också mikrokontroller och andra datorer blivit förmånligare. Eftersom inbyggda system har blivit vanligare och förmånligare har förväntningarna på inbyggda system också stigit, dels till följd av dess breda användningsområde, men också för att de ofta hanterar känslig och privat data [1].

I kritiska tillämpningar är det viktigt att den inbyggda enheten är stabil under drifttid eftersom problem i ett kritiskt system vid fel ögonblick kan ha allvarliga konsekvenser. Eftersom systemen är en del av andra apparater är de svåra att komma åt och i många fall är det viktigt att de fungerar felfritt under längre tider i krävande förhållanden utan underhåll [4].

2.1 Utmaningar vid utveckling

Det finns många utmaningar vid utveckling av program för inbyggda system. En av de största utmaningarna är att systemen ofta har begränsade resurser på grund av storleken på datorerna. Det här gör att processorerna har låg prestanda, lagringsutrymmet och minnet är litet, och de försörjs ofta med små mängder ström. Inbyggda system måste därför planeras noggrant för att utnyttja resurserna så effektivt som möjligt. För en så effektiv användning av komponenterna som möjligt krävs att koden skrivs på en lägre nivå för att ha bättre kontroll över systemets resurser. Komponenterna måste också väljas utifrån specifikationerna på systemet vilket ofta leder till att storleken på den inbyggda datorn är liten och saknar många funktioner som t.ex. tillgång till ett grafiskt gränssnitt eller nätverkskommunikation [4]. Saknaden av trådlös kommunikation till den inbyggda datorn förhindrar även smidiga uppdateringsmöjligheter och det är svårare att övervaka systemen. Det här gör betydelsen för en välfungerande mjukvara i systemet allt större.

En annan utmaning vid utvecklingen är att systemen måste klara av att prestera i realtid, exempelvis i system som läser och bearbetar data från sensorer för att övervaka större system. Ett system som inte klarar av att prestera kan ge oväntade och farliga konsekvenser. Det här gör det viktigt att anpassa allt från arkitekturen på systemet till själva programmet. Programmet måste optimeras för att klara av att utföra uppgifter med rätt hastighet och samtidigt garantera att det inte förekommer oväntade fel och buggar som stör systemet under drifttid.

Slutligen kräver ofta utvecklingen av inbyggda system specialiserad kunskap inom området där de tillämpas. Till exempel när ett inbyggt system för ett flygplan implementeras måste utvecklaren förstå sig på teknologierna som används och vilka konsekvenser ett fel kan leda till. I somliga tillämpningar kräver också programmeringsspråken certifikat för att uppnå standarden för systemet [3].

3 Introduktion till Rust

Rust är ett modernt programmeringsspråk som är gjort för att förse programmeraren med minnesäkerhet, säker användning av flera trådar och effektivitet, och ska samtidigt vara lätt att använda. Kontroll över resurser på en låg nivå, med abstraktioner för att göra utvecklingen smidigare är en viktig del av språket Rust. Språket lanserades år 2010 av företaget Mozilla och har blivit populärt bland utvecklare bland annat för att

bygga servrar och för utveckling av Linuxkärnan [6]. Rust har visat sig vara det mest omtyckta programmeringsspråket bland utvecklare sedan år 2016 enligt frågeformulär på webbplatsen Stack Overflow [7]. Sedan år 2018 har ett av de centrala målen för Rust även varit att och utveckla språket och verktygen runt omkring för att bli smidigare och bättre för utveckling av inbyggda system [3].

Språket Rust är inspirerat av flera andra programmeringsspråk och har därför en syntax som liknar många andras. För att skilja sig från de andra har Rust bestämt flera regler för att vara strängare vid hanteringen av data för att ge mer kontroll över programmen och högre säkerhet i den slutliga produkten. Som andra moderna programmeringsspråk har Rust också ett brett typsystem och är försett med en kompilator och en pakethanterare för att göra utvecklingen smidigare. En kompilator är en programvara som används för att omvandla skriven kod till maskinkod som datorer klarar av att läsa.

I följande avsnitt presenteras programmeringsspråket Rust och dess användningsområden. Språket jämförs också andra programmeringsspråk, t.ex. C som ofta används i inbyggda system.

3.1 Variabler, datatyper och funktioner

I Rust definieras en variabel med nyckelordet ”let” följt av variabelns namn, ibland typ och sist värdet. Om ingen typ specificeras för en variabel kommer typen bestämmas automatiskt för att passa värdet för variabeln. Variabler i Rust stöder enkla typer som heltal, booleska värden, flyttal, strängar och även mer komplexa typer som räckor och strukturer (eng. struct). Värden i Rust har en bestämd typ och kan på så sätt inte bytas från en typ till en annan [8]. Exemplet nedan visar hur variabler definieras i Rust.

```rust
let = 42;            // saknar typ -> 32 bitars heltal
let y: f64 = 3.14;   // f64 -> 64 bitars flyttal
let z: bool = true; // bool -> booelsk variabel
```

Funktioner i Rust definieras med nyckelordet ”fn” följt av funktionens namn och parenteser. Parenteserna kan innehålla funktionsargument av olika typer. Funktioner tillåts även som funktionsargument. Efter parenteserna följer en frivillig syntax med en pil följt av en datatyp. Pilen indikerar vilken typ av data som funktionen returnerar efter ett anrop. Funktioner kan även returnera funktioner. Det här betyder att Rust kan användas som ett funktionellt programmeringsspråk, där programmet byggs upp av en rad funktionsanrop

för att kontrollera flödet i programmet [8]. En funktion som adderar två heltal och returnerar summan visas nedan.

```
fn addition(x: i32, y: i32) -> i32 {
    x + y // Funktionen addera x och y och returnera summan
}
```

Rust har även avancerade typer som strukturer och enumerationer. En struktur i Rust är en representation av data i en grupperad form och betecknas med nyckelordet ”struct”. Strukturer kan innehålla alla datatyperna, även andra strukturer. Strukturer kan även implementera egna funktioner, som kallas metoder. Metoderna kan endast nås genom instanser av den specifika strukturen. En instans är en specifik förekomst av någon datatyp. Metoderna ger Rust objektorienterade egenskaper. Nyckelordet för implementation är ”impl”. Exemplet nedan visar hur syntaxen för strukturer och implementationer av metoder ser ut [3][8].

```
struct Vektor2D { // Stukturen Vektor2D innehåller flera attribut
    x: f32,       // x representerar x-koordinaten
    y: f32,       // y representerar y-koordinaten
}
impl Vektor2D {                  // implementation av strukturen
    fn storlek(&self) -> f32 { // Referens till vektorn som argument
        (self.x * self.x + self.y * self.y).sqrt()
    }   // Returnerar längden av vektorn
}
fn main() {
    let vektor = Vektor2D { x: 6.0, y: 8.0 }; // Vektor med x=6, y=8
    println!("Längd = {}", vektor.storlek()); // printar Längd = 10
}
```

Det är även möjligt att bygga modeller som strukturer kan implementera, dessa kallas egenskaper (eng. trait). Då en struktur implementerar en egenskap ger den instanser av strukturen tillgång till alla funktionerna som egenskapen förväntas innehålla. Egenskaper kan innehålla funktioner som inte är färdigt implementerade och då måste strukturer implementera de funktionerna. Till exempel får en struktur *Hund* som implementerar egenskapen *Djur* tillgång till alla funktionerna som är en del av egenskapen *Djur*. De funktionerna som inte är implementerade i egenskapen *Djur* fungerar som en mall av en funktion som måste implementeras av strukturen *Djur* för att strukturen ska uppfylla

kraven som ställs av egenskapen [8].

```
trait Djur { // Definiera en egenskap "Djur"
    fn vikt(&self); // "vikt" måste implementeras av strukturer
    fn spring(&self) { // "spring" är en del av egenskapen
        println!("Djuret springer!");
    }
}
struct Hund { // Definiera en struktur "Hund"
    vikt: i32,
}
impl Djur for Hund { // Implementera egenskapen hos en struktur
    fn vikt(&self) {
        println!("Hunden väger {} kg", self.vikt);
    }
}
fn main() { // Anropa metoden "vikt" genom en instans av strukturen
    let hund = Hund { vikt: 21 };
    hund.vikt(); // Printar "Hunden väger 21 kg"
}
```

En enumeration är en uppräknelig datatyp som betecknas med nyckelordet ”enum”. Enumerationer innehåller namngivna värden, som kallas varianter. Varianterna kan också associeras med en datatyp. De används typiskt för representation av olika alternativ, exempelvis för att representera veckodagarna eller olika resultat [8]. Enumerationer används också ofta för att beskriva olika feltyper för att ge en mer detaljerad beskrivning då problem uppstår, vilket betyder att de kan fungera som felkoder i program [9].

```
enum Resultat {     // Enumerationen Resultat definieras
    Ok,             // Värdet Ok
    Fel(String),    // Värdet Fel - innehåller en sträng
}
// vitsord är ett 8 bitars osignerat heltal
fn tent_resultat(vitsord: u8) -> Resultat {
    if vitsord > 0 { // Returnera Ok om vitsord > 0
        Resultat::Ok
    } else { // Annars fel med en beskrivning
        Resultat::Fel(String::from("Poängen räckte inte till"))
    }
```

```
}
fn main() {
    match tent_resultat(0) { // Kontrollerar alla möjliga resultat
        Resultat::Ok => println!("Godkänd!"),
        Resultat::Fel(svar) => println!("Underkänd: {}!", svar),
    } // Programmet printar "Underkänd: Poängen räckte inte till!"
}
```

3.2 Ägarskapsmodellen

Rust är ett statiskt skrivet programmeringsspråk. Ett statiskt skrivet programmeringsspråk innebär att typerna av variabler eller annan data, exempelvis strängar, kontrolleras i samband med kompileringen, istället för att kontrolleras under drifttid. Kompilering kan beskrivas som byggandet av koden. I Rust konverteras koden från skriven kod till maskinkod då den kompileras. Maskinkod är kod på en lägre nivå som består av instruktioner som datorer kan följa. Att Rust är statiskt innebär att fel i koden upptäcks i ett tidigt skede eftersom programmet inte kommer kompileras om det finns ett eller flera fel i koden [2][8].

Rust garanterar minnessäkerhet vid kompileringen, vilket minskar sannolikheten för att många typer av misstag uppstår då programmet byggs. Minnessäkerheten uppnås av en ägarskapsmodell som grundar sig på att varje värde endast har en ägare vid en viss tidpunkt [9]. Ägaren har tillgång till värdet så länge som värdet är innanför räckvidden i koden. Då räckvidden inte räcker till, exempelvis då en funktion återvänder, kommer ägaren direkt frigöra minnet och värdet är inte längre användbart. Samma princip följs då värdet byter ägare. Byte av ägare i Rust sker om man flyttar över en variabel till en annan. Då ett värde byter ägare kommer den ursprungliga ägaren inte vara användbar längre och minnet frigörs direkt, vilket förhindrar minnesläckor. I program med flera trådar och processer förhindrar det oväntade förändringar i värdet [3].

Ägarskapsmodellen möjliggör att minneshanteringen i Rust är säker och sker helt automatiskt även om den inte kräver ett sophanteringssystem som körs i en skild process och övervakar vilka variabler som är användbara. Ett sophanteringssytem är en mekanism som kontrollerar om värden fortfarande är användbara och då värdet inte är användbar frigör sophanteraren automatiskt minnet som värdet använder. Ett sophanteringssystem kräver resurser av datorn för att uppnå minnessäkerhet i programmet. Att inte ha ett system för hanteringen av minne gör att Rust kan utnyttja resurserna effektivare. Det här

är en av anledningarna till att Rust kan uppnå samma hastigheter som språk med manuell sophantering, där programmeraren är ansvarig över frigörande av minne. Ett exempel på ägarskap kan vara överföringen av värdet från en variabel till en annan, och senare försöka nå värdet genom den ursprungliga ägaren, som redan frigjort minnet, vilket kompilatorn upptäcker och koden kompileras inte. Exemplet nedan visar hur detta kan se ut.

```
fn main() {
    let x = String::from("Hej!");  // Variabel x äger värdet "Hej!"
    let y = x;  // y tar över ägarskapet från x
    // x är inte längre användbar eftersom värdet ägs av y
    println!("x = {x}, y = {y}");  // Detta komplierar inte
}
```

Utlåning av ett värde kan göras genom att definiera en variabel med en referens till den ursprungliga ägaren. Det här möjliggör att flera variabler kan dela samma värde. Utlåning följer stränga regler definierade av en lånekontrollerare (eng. borrow checker). Lånekontrolleraren är gjord för att garantera att varje har endast en ägare och för att förhindra fel i program med samtidiga trådar som har åtkomst till samma värden. Det här innebär också att en variabel med referens till en annan ägares värde inte har möjligheten att förändra värdet. Referenser till värden betecknas med en ampersand, se exemplet nedan. Typer kan också implementera egenskapen *Copy*, som gör att värden inte flyttas till en ny ägare utan istället kopieras värdet och den nya variabeln är en kopia med en egen minnesadress. [8].

```
fn main() {
    let x = String::from("Hej!");; // Variabeln x äger värdet 5
    let y = &x;                    // y är en referens till x
    println!("x = {x}, y = {y}");  // Printar "x = Hej!, y = Hej!"
}
```

Som standard följer Rust många regler för att göra koden så säker som möjligt. Det går att överskrida säkerheten genom att aktivera osäkert Rust (eng. unsafe Rust) [8]. Den osäkra varianten tillåter interaktion med kod skriven i andra språk, som t.ex. C, vilket medför nya risker i programmen. Den här delen av Rust kan kompilatorn inte heller fullständigt granska, vilket ökar risken för minnesläckor och andra säkerhetsbrister. Mindre än 30% av biblioteken som är tillgängliga för Rust kräver denna egenskap. Däremot finns många bibliotek där kompilatorn inte fullständigt klarar av att statiskt analysera hela koden, vilket gör det utmanande för Rust att verkligen garantera typ- och minnerssäkerhet. Eftersom

inbyggda system har dominerats av programmering i språket C, används ofta färdig C-kod eller operativsystem byggda med C i Rust-applikationer för inbyggda system [3][9].

3.3 Föränderlighet

Som standard går det inte att ändra värdet för variabler i Rust, utan det värde som anges kommer förbli oförändrat. Detta är en egenskap för att förhindra oförutsägbar kod och göra programmeraren mer medveten om vilka värden som faktiskt kommer förändras. För att skapa värden som går att förändra kan man använda nyckelordet ”mut” för namnet på värdet. Även om man skapar ett värde som går att förändra kan typen på värdet aldrig ändras. Exemplet nedan visar hur man kan skapa och modifiera variabler [8].

```
fn main() {
    let mut x = 5;          // föränderlig variabel x med värdet 5
    println!("x = {}", x);  // Printar "x = 5"
    x = 15;                 // värdet på x förändras till 15
    println!("x = {}", x);  // Printar "x = 15"
}
```

3.4 Felhantering

I programmering är felhantering viktigt för att garantera att program gör vad som förväntas och att program inte plötsligt slutar att fungera. I Rust måste alla värden hanteras på något sätt, detta gäller också fel. Felen i Rust är som andra datatyper så de kan vara en del av returvärdet från en funktion. I de fall att programmet inte kan återhämta sig från ett fel innehåller Rust funktionen ”panic” som kommer att avsluta exekveringen av programmet omedelbart. I de fall att programmet kan återhämta sig från ett fel har Rust även funktioner för att hantera felen på ett varsamt sätt. Felen som produceras av funktioner i Rust returneras typiskt som en del av typerna ”Option” eller ”Result”. Anpassade typer av fel är ofta definierade som enumerationer, vilket hjälper att urskilja olika typer av fel från varandra [8].

3.5 Parallellism

Parallellism inom programmering betyder att flera uppgifter utförs av programmet samtidigt i olika trådar. Det här utnyttjas ofta för att försnabba exekveringen genom att

dela upp utförandet av uppgifter till flera av processorns kärnor. Det kan också vara kritiskt för program att utföra flera uppgifter samtidigt för att inte gå miste om viktig information från olika delar av programmet. Det finns flera risker i program som utför flera uppgifter samtidigt. Ett vanligt fel som kan uppstå är att synkroniseringen mellan trådarna inte fungerar. Fel i synktroniseringen kan leda till att programmet producerar fel data eftersom en eller flera trådar inte har exekverat färdigt innan utdatan blivit producerad. I inbyggda system är det viktigt med samtidiga uppgifter i program, eftersom de möjliggör funktioner som läsning av data från flera sensorer eller andra enheter samtidigt [4].

Som ett mer modernt programmeringsspråk har Rust fördelen att se hur andra språk gått tillväga vid användning av flera trådar. Det här har gett Rust ett system som inte kräver manuell synkroniserning mellan trådarna [10]. Dessutom förstår kompilatorn i Rust vilka värden som delas mellan olika uppgifter och på så sätt är även denna funktion kontrollerad av en kompilator för att hitta fel i hanteringen av minne, ägarskapet av värden och den kan förhindra kapplöpning. Kapplöpning sker när flera delar av koden försöker nå samma värde vid samma tidpunkt. Rust beskrivs därför som ett språk där flera samtidiga uppgifter kan utföras utan rädsla [3].

I jämförelse med språket C ger detta Rust en stor fördel eftersom C kräver interaktion med systemet för att skapa flera trådar. Då flera uppgifter utförs i C måste koden skrivas på rätt sätt för att förhindra kapplöpning och andra fel. Kompilatorer för C upptäcker inte heller fel i koden mellan flera trådar. C kräver också att programmeraren ser till att synkroniseringen av fler trådar sker på rätt sätt.

3.6 Jämförelse mellan Rust och andra språk

Eftersom C är det mest använda språket i inbyggda system kan det användas för jämförelse för att förstå hur andra språk kan fungera i inbyggda system [7]. Det är även viktigt att jämföra Rust med andra språk för att förstå hur språket står upp mot andra moderna språk [2].

3.6.1 Programmeringsparadigmer

Programmeringsparadigmer används för att beskriva och separera olika typer av programmeringsstilar [3]. De flesta språken inkluderar möjligheten till användning av flera paradigmer vid utvecklingen. I både Rust och C förkommer paradigmerna

strukturerad programmering, imperativ programmering, procedurell programmering, funktionell programmering och former av parallell programmering. I Rust kan man uppnå objektorienterad programmering genom att implementera metoder för strukturer, vilket inte är möjligt i C. Strukturerad programmering är den vanligaste paradigmen och innebär att exekveringen av koden sker i sekvenser där flödet kan ändras genom olika uttryck, exempelvis genom if-satser. C har flera uttryck som kan användas för att uppnå detta men båda språken följer principerna för paradigmen. Imperativ programmering betyder att uttryck stegvis förklarar hur koden ska exekveras, vilket både C och Rust gör. Procedurell programmering innebär att program implementerar procedurer som kan kallas på och återanvändas i koden. Rust och C har till exempel funktioner som gör detta möjligt. Funktionell programmering är programmering där programmen byggs som ett träd av funktionsanrop. Funktionell programmering innehåller ofta anonyma funktioner som liknar normala funktioner men kräver inte ett namn och kan vara definierade som uttryck istället för fullständiga funktioner. I Rust är detta möjligt eftersom funktioner kan ges som argument och Rust också har anonyma funktioner. I C är det möjligt att uppnå funktionell programmering, men C har inga anonyma funktioner och funktioner kan inte vara ett funktionsargument. Parallell programmering, eller programmering med flera trådar, innebär att programmet kan exekvera kod samtidigt i flera trådar och på så sätt utföra olika funktioner samtidigt. C uppnår detta genom att använda egenskaper hos operativsystemet även om det inte är en del av programmeringsspråket. Detta leder också till att synkroniseringen mellan trådarna inte är ett måste vilket kan leda till oväntade resultat. I Rust är samtidig programmering en av de starkaste sidorna både på grund av säker datahantering mellan trådar och kompilatorn som granskar säkerheten vid kompilering av programmet.

Strukturen hos språken C och Rust liknar varandra, men Rust är ett modernare språk som för med sig möjligheten för objektorienterad utveckling och även säkerhet då flera trådar används av samma program. Säkerheten kan vara till stor nytta i kritiska system [9].

3.6.2 Produktivitet och verktyg

För att skapa ett program av hög kvalitet krävs erfarenhet och noggrannhet. Kvaliteten och produktiviteten kan stärkas med hjälp av olika verktyg som kompilatorer, bibliotek för testning av kod och pakethanterare. Då det handlar om att bygga inbyggda system måste biblioteken vara tillämpade för samverkande med datorer på en lägre nivå och klara av att hantera olika typer av data, som t.ex. inkommande analoga signaler från sensorer. Rust är försett med många bibliotek och verktyg som hjälper att effektivera utvecklingen

av inbyggda system. Kompilatorn i Rust är även strikt och granskar koden noggrant för att upptäcka fel. C har en mängd olika kompilatorer varav de vanligaste är GCC och Clang. C-kompilatorer är inte minnessäkra och ger inte heller lika tydliga felmeddelanden som kompilatorn i Rust [3].

Tester av kod kan hjälpa till både med att validera att delar av programmet fungerar enligt förväntningar och de kan även förhindra att fel uppstår av misstag vid uppdateringar av koden. C har flera tillgängliga bibliotek som är gjorda för testning av koden, till exempel Unity, som är gjort specifikt för tester av inbyggda system. Rust innehåller en standard modul för testning som är väl dokumenterade och kan användas till de flesta typerna av enhetstester. Det förekommer även bibliotek gjorda för att utvidga testegenskaperna vid utveckling av inbyggda system [3].

Rust har en pakethanterare som fungerar smidigt och paketen är centrerade till en och samma plats. Det här gör det smidigt att hitta färdig kod som kan användas vid utveckling. C-paket kräver i regel konfigurationer och en stor del av paketen är inte fritt tillgängliga. Pakethanterare har inte blivit en viktig del av C vilket påverkar smidigheten vid utvecklingen. På grund av att C är äldre och mer använt finns flera tillgängliga verktyg som har blivit använda under många år och därför är det ofta enklare att lösa ett problem i C än i Rust även om Rust har centraliserat paketen till en och samma plats.

Verktyg möjliggör att kod kan skrivas snabbare och fungera på flera system. Om man jämför kompilatorn i Rust med andra programmeringsspråk finns en betydande likhet, nämligen LLVM (Low-Level Virtual Machine). Rusts kompilator stärks av LLVM, som är en samling av teknologier för kompilatorer för att hjälpa konvertera skriven kod till maskinkod. LLVM används också i kompilatorer för andra programmeringsspråk, exempelvis i Clang för språket C. LLVM kan kompilera kod för att fungera på en mängd olika system med olika arkitekturer, vilket gör det enklare att bygga stöd för Rust på flera system. LLVM stöder inte alla arkitekturer. I C kan man t.ex. utnyttja andra kompilatorer för att rikta in sig på system där LLVM inte fungerar. Eftersom Rust endast är försett med en kompilator som använder sig av LLVM begränsar det tillämpningsområdena [3].

Förutom hjälp av kompilatorn hjälper operativsystem programmerare att skriva kod utan att kräva direkt kontroll av resurserna. I inbyggda system är det ofta omöjligt att använda operativsystem som Linux på grund av begränsade resurser. Det är vanligt att istället använda realtidsoperativsystem (RTOS), vilka är gjorda för att effektivt utnyttja system med mindre resurser [3]. Problemet med RTOS är att de alla är skrivna med programmeringsspråket C. Oberoende om Rustprogram som använder RTOS är minnessäkra garanterar operativsystemet inte minnessäkerhet. Förutom RTOS för

inbyggda system finns operativsystem skrivna i Rust. Tock är ett operativsystem skrivet i Rust och är riktat för inbyggda enheter. Tock garanterar minnessäkerhet och är byggt för att uppfylla krav hos kritiska system [11].

3.6.3 Hastighet och effektivitet

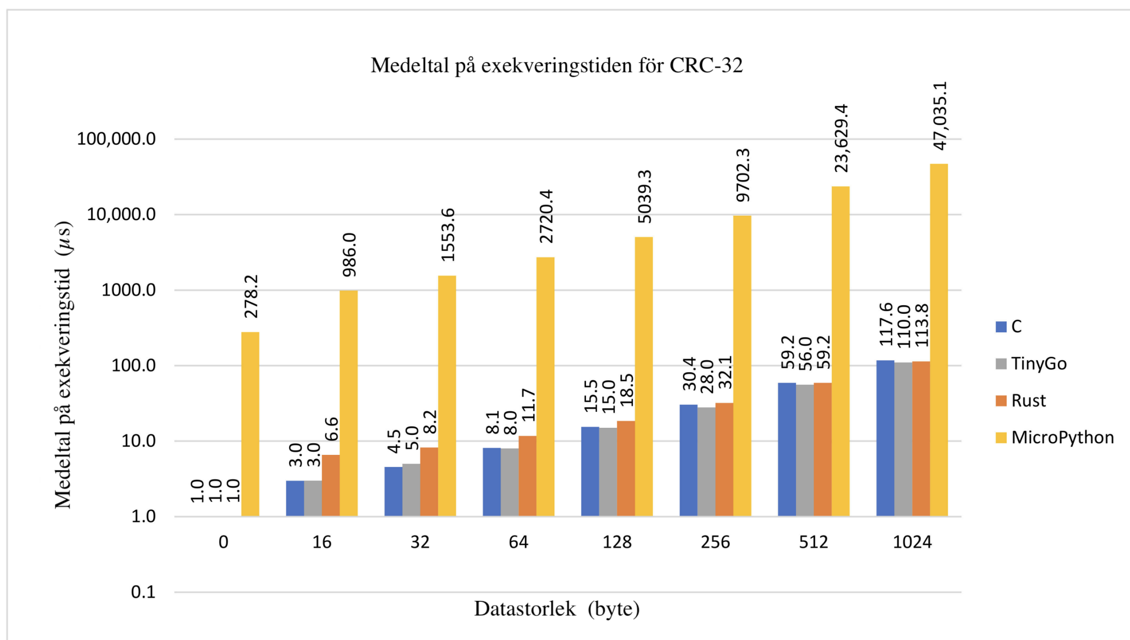
Det har blivit vanligare att fundera på energikonsumptionen för nya datorer och deras tillämpningar. Både i tillverkningen av hårdvara och i utvecklingen av mjukvara har det blivit allt viktigare att förstå energiförbrukningen för den slutliga produkten. Mjukvaruutvecklare har samtidigt blivit noggrannare med att optimera koden och de undviker programmeringsspråk som kräver mycket energi när det är möjligt [12]. För att minimera energiförbrukningen är det förutom att optimera koden även viktigt att välja det rätta programmeringsspråket. Ett programmeringspråk med högre prestanda kräver mindre resurser. Generellt kräver också snabbare programmeringsspråk mindre energi för att utföra funktioner [13]. Bland de 27 mest använda programmeringsspråken är Rust rankat på andra plats efter C både i exekveringstid och energikonsumption. Tabell 3.1 visar resultaten för alla 27 programmeringsspråken.

	Energi
C	1.00
Rust	1.03
C++	1.34
Ada	1.70
Java	1.98
Pascal	2.14
Chapel	2.18
Lisp	2.27
Ocaml	2.40
Fortran	2.52
Swift	2.79
Haskell	3.10
C#	3.14
Go	3.23
Dart	3.83
F#	4.13
JavaScript	4.45
Racket	7.91
TypeScript	21.50
Hack	24.02
PHP	29.30
Erlang	42.23
Lua	45.98
Jruby	46.54
Ruby	69.91
Python	75.88
Perl	79.58

	Tid
C	1.00
Rust	1.04
C++	1.56
Ada	1.85
Java	1.89
Chapel	2.14
Go	2.83
Pascal	3.02
Ocaml	3.09
C#	3.14
Lisp	3.40
Haskell	3.55
Swift	4.20
Fortran	4.20
F#	6.30
JavaScript	6.52
Dart	6.67
Racket	11.27
Hack	26.99
PHP	27.64
Erlang	36.71
Jruby	43.44
TypeScript	46.20
Ruby	59.34
Perl	65.79
Python	71.90
Lua	82.91

Tabell 3.1: Programmeringsspråk ordnade enligt energiförbrukning och tidsanvändning. Kortare tid och lägre energiförbrukning är bättre. Tabellen är modifierad från [13].

I inbyggda system kan energiförbrukningen och effektiviteten se annorlunda ut eftersom arkitekturen och komponenterna i datorerna varierar mycket beroende på användningsområdet. ESP32 är en mikrokontroller som är populär bland utvecklare för dess breda användningsområde, låga pris och stöd för mjukvara skriven i flera olika programmeringsspråk. För att få en bild över hur Rust står upp mot andra programmeringsspråk som kan utnyttjas i inbyggda system kan ESP32 mikrokontrollern användas som en enhet för att göra olika tester. Beräkningar av kontrollsummor av typen CRC-32 (figur 3.1) och filter, samt utförande av kryptografiska algoritmer utförda på en ESP32 mikrokontroller i programmeringsspråken C, Rust, MicroPython och TinyGo visar hur olika programmeringsspråk presterar på system med mindre resurser [14]. TinyGo och MicroPython är lättare versioner av programmeringspråken Go och Python lämpade specifikt för hårdvara som ofta förekommer i inbyggda enheter. Testerna ger två viktiga resultat. Det första är att programmeringsspråket TinyGo normalt är långsammare än både Rust och C, men i testerna har TinyGo inget operativsystem och använder direkt systemets resurser. De tre andra programmeringsspråken har någon typ av operativsystem mellan programmet och hårdvaran. I detta fall klarar därför TinyGo av att prestera bäst. Det andra viktiga resultatet är att prestandan för Rust i inbyggda system ligger väldigt nära programmeringsspråket C.



Figur 3.1: Medeltal av exekveringstiden för kalkyl av CRC-32 kontrollsummor utfört av programmeringsspråken C, TinyGo, Rust och MicroPython. Testerna är utförda på en ESP32 med olika storleker på datan. Resultatet anges i mikrosekunder och skalan är logaritmisk. Bilden är modifierad från [14].

4 Rust i inbyggda system

Rust är ett programmeringsspråk som skiljer sig från andra genom att hålla sig till strikta regler, även om språkets syntax liknar många andra programmeringsspråk. Rust är även snabbt och kräver inte mycket energi vilket kan göra det till ett utmärkt alternativ vid programmeringen av inbyggda system. I följande avsnitt presenteras fördelar och nackdelar med Rust i inbyggda system, samt förbättringar som kan göra Rust ett mer optimalt programmeringsspråk för inbyggda system.

4.1 Fördelar

4.1.1 Säkerhet

Minnes- och typsäkerhet, en garanterad prestanda och pålitlighet är de främsta egenskaperna hos Rust. I inbyggda enheter kan det här begränsa olika problem och buggar eftersom programmet inte alls byggs i de fall att koden inte uppfyller alla krav som kompilatorn ställer.

En buffert i en dator är en temporär lagringsplats i datorns minne och används vid flöde av information mellan olika processer. Om en process skriver mer data till en buffert än den kan innehålla sker en överfyllning. Detta leder till att datan skrivs på en oväntad plats och kan orsaka problem som att programmet stannar eller filer blir korrupta. Buffertöverfyllning är en konsekvens som kan vara minnesrelaterad och uppstå i samband med hantering av minne på fel sätt. Rust förhindrar minnesreleterade fel genom att granska hanteringen av minne i samband med att programmet kompilerar. Osäkert Rust är möjligt att använda för att kringgå minneskontroller, men som standard kommer kompilatorn granska hela programmet [1][4].

Avreferering av nullpekare innebär att ett program försöker använda ett värde som inte är giltigt. En nullpekare är en pekare som inte pekar på någon minnesadress. I Rust har detta förhindrats genom att tvinga programmeraren att hantera alla värden. Om ett värde inte kontrolleras kommer kompilatorn inte bygga programmet. Användning av variabler eller annan data efter frigörande av det associerade minnet är också förbjudet vilket också resulterar i att programmet inte byggs [1].

I inbyggda system det viktigt att klara av att utföra flera uppgifter samtidigt. Både avreferering av nullpekare och minneshanteringen är säker i program skrivna i Rust

som exekverar flera delar av koden samtidigt. Det här ger Rust en stor fördel i jämförelse med C.

4.1.2 Användarvänlighet

Användarvänligheten hos ett programmeringsspråk beror på flera aspekter såsom dokumentation, användarmanualer och hjälp av andra användare. Rust har centraliserat dokumentationen på deras hemsida så den är mer lättillgänglig. En bok om programmeringsspråket Rust finns skriven och är öppen för alla [8]. Rust har även en bok som handlar om inbyggda system och en för hur utvecklar av bibliotek förväntas skriva dokumentation. Boken om inbyggda system ger utvecklare en bra start för koncept och struktur för projekt riktade till inbyggda enheter. Boken om dokumentation är även viktig, eftersom den försöker få utvecklare att både skriva dokumentation och att dokumentation gjorda av olika utvecklare ska följa samma stil. De här faktorerna gör Rust till ett mer omtyckt och lättillgängligt programmeringsspråk [3].

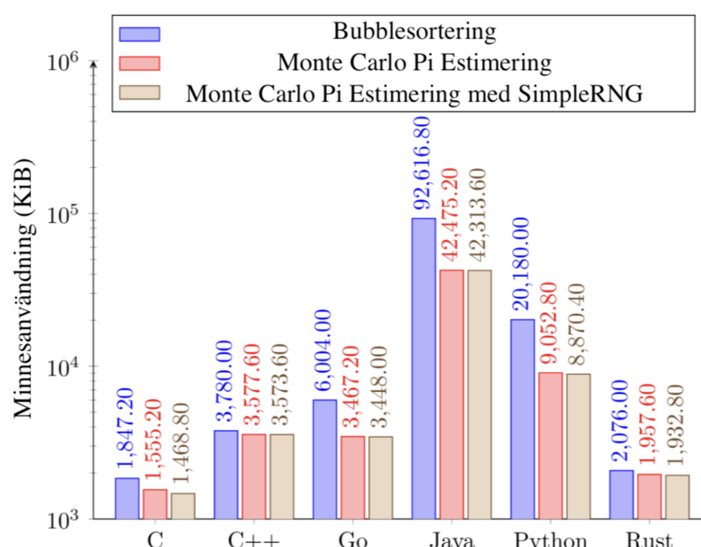
Gemenskapen inom Rust är viktig för att utvecklare ska få hjälp med problem de stöter på och få dela nya idéer med andra utvecklare. En stor del av Rust gemenskapen finns på hemsidan Github. Rust har även grupper av utvecklare som specialiserats på olika programmeringsområden. En av grupperna kallas *Embedded Devices Working Group* som har uppgiften att både utveckla språket och verktygen för inbyggda system [3].

Kompilatorn i Rust är också gjord för att hjälpa utvecklaren förstå var de fel som kompilatorn upptäcker finns och vad som är orsaken till felen. Ledtrådar till möjliga korrigeringar visas också för utvecklaren. Ett exempel där kompilatorn visar hur ett värde har bytt ägare och senare försökts användas genom den ursprungliga ägaren syns nedan.

```
error[E0382]: borrow of moved value: ‘x‘.  --> src/main.rs:5:20
2| let x = String::from("Hej!");
 |      - move occurs because ‘x‘ has type ‘String‘, which does not
 |        implement the ‘Copy‘ trait
3| let y = x;
 |         ^ value moved here
5| println!("x = {x}, y = {y}");
 |                   ^ value borrowed here after move
help: consider cloning the value if performance cost is acceptable
```

4.1.3 Prestanda

Rust uppnår en prestanda som kan jämföras med C. Jämfört med andra moderna som nått en bred utspridning är Rust ofta överlägsen. För inbyggda system som måste fungera i krävande förhållanden och klara av att utföra uppgifter i realtid är det här en stor fördel [3]. Under drifttid är det även viktigt att programmet har tillräckligt med resurser. Minneshantering måste därför ske effektivt för att minnet inte ska bli en flaskhals för programmet. En studie visar minnesanvändningen vid utförandet av olika algoritmer för bland annat programmeringsspråken Rust, C och Go (figur 4.1) [7]. Rust rankas liksom för hastigheten och energianvändningen igen på en andra plats efter språket C i minnesanvändningen.



Figur 4.1: Medeltal på minnesanvändingnen vid tre olika algoritmer för programmeringsspråken C, C++, Go, Java, Python och Rust. Minnesanvändningen är given i kilobyten. Figuren är modifierad från [7].

4.2 Utmaningar och begränsningar

4.2.1 Begränsade plattformer och verktyg

Rust kan utnyttja egenskaper hos LLVM för att kompilera kod för att fungera på flera plattformer, men det kräver att LLVM stöder plattformen. På flera arkitekturer kan RTOS användas för att kontrollera funktioner och resurser för systemet. RTOS-implementationer garanterar inte full säkerhet. På flera plattformer som Rust stöder begränsas också

tillämpningen av Rust på grund av att att de endast är delvis implementerade. Det här begränsar olika funktioner och gör att utvecklare som använder Rust måste välja enheten efter vad Rust stöder [3].

Rust har många funktioner som kan utnyttja operativsystemet för att samverka med hårdvaran. Om systemet kräver program utan operativsystem finns en nerskalad version av Rust som utesluter funktioner gjorda för interaktion med operativsystem [15]. Eftersom denna del av Rust inte har alla inbyggda funktioner, som hantering av filer, är utvecklingen också svårare. Tock fungerar som ett verktyg för att använda Rust utan operativsystem, men här igen begränsas användningen av få fungerande plattformer. Tock saknar även egenskaper som utförande av uppgifter i realtid, vilket är viktigt i system som kräver låg responstid [9].

Verktygen för inbyggda system visar sig också vara en begränsning, speciellt då inbyggda systemen utvecklas i professionellt syfte. En orsak till begränsningen är att ofta saknar certifikat som krävs i vissa tillämpningar. Även om Rust uppfyller krav för många certifikat som C också uppfyller, är föredras ofta C på grund av dess tidigare användning i många kritiska tillämpningar. Rusts kompilator saknar också certifikat. I Rust sker dessutom mer förändringar än i gamla språk och det ses därför som ett mindre stabilt programmeringsspråk [3].

4.2.2 Storleken på programmen

En stor likhet mellan Rust och andra språk som tillämpas vid inbyggda system är att koden byggs till exekverbara filer som kallas binärer. Studier visar att Rust-binärer är 79% större än motsvarande binärer gjorda i C för tillämpningar i system med begränsade resurser [16]. Den stora storleken orsakas i främst av egenskaper som gör Rust ett bekvämt programmeringsspråk, t.ex. egenskaper som objektorientrad programmering, en detaljerad felhantering och automatiskt genererad kod vid kompileringen. Genom att exempelvis undvika en detaljerad felhantering kan storleken på programmen minska, men samtidigt gör det utvecklingsprocessen och övervakningen av programmen mindre smidig. Det går också att undvika statiska analyser kring minneshanteringen genom att använda osäkra pekare. Det här minskar storleken på binärerna, men samtidigt strider det mot minnessäkerheten och gör Rust mera likt programmeringsspråket C.

4.3 Förbättringar

Nackdelarna hos Rust är inte många. De flesta nackdelarna finns på grund av en begränsad tillgång till färdigt utvecklade verktyg. Vid utvecklingen av inbyggda system finns ytterligare två faktorer som påverkar användningen av Rust. Den första är att Rustkod inte stöds av all hårdvara och den andra är storleken på programmen. För att göra språket optimalt för flera inbyggda system krävs alltså förbättringar inom flera områden.

Bibliotek och paket specificerade för inbyggda enheter kräver ett högt fokus för att Rust ska bli det mest optimala språket. För tillfället saknas många bibliotek, vilket gör att programmerar måste bygga dessa själv. Genom att göra nya verktyg fritt tillgängliga och dokumentera implementationerna enligt Rusts förväntningar kan en enskild utvecklare hjälpa språket bli bättre för utvecklade system. För att vidare göra språket mer utvecklat är det viktigt att nya bibliotek upprätthålls, vidareutvecklas och korrigeras då möjliga fel upptäcks. Utvecklingen och upprätthållningen gäller även för nuvarande bibliotek, främst de som fortfarande inte uppnåt versionen 1.0 [3].

Storleken på binära filer går att minska genom att minimera användningen av olika egenskaper som Rust innehåller. Även om vissa egenskaper, t.ex. objektorienterad programmering, gör utvecklingen smidigare, kan storleken på programmen växa då varje implementation genererar sin egna kod. Genom att istället manuellt kontrollera typen kan man reducera stoleken på programmet. Det visar sig att alla typer som använder sig av funktionen ”panic” vid fel som går att återhämtas från innehåller ett meddelande för att beskriva felet. Då programmet kompileras kommer både funktionen och meddelandet följa med alla typer som använder denna. Genom att istället utföra en mer grundlig felhantering och undvika nyckelordet ”panic” kan man minska kring 10% av storleken på programmet [16].

De svåraste begränsningarna att överkomma för Rust i inbyggda system är att språket inte går att kompilera till lika många plattformer som mera utvecklade programmeringsspråk. För att uppnå en bredare utspridning inom inbyggda system måste Rust därför utveckla verktyg och egenskaper för att vara tillgängligt för flera typer av arkitektur på hårdvaran. Eftersom kompilatorn använder sig av LLVM kan Rust utnyttja denna för att kompilera program för arkitekturer som redan stöds av LLVM. Då Rust utvecklas för att stöda fler plattformer är det viktigt att koden håller sig till säker minneshantering och minimerar användningen av kod som genererar större program.

5 Diskussion

Rust är ett modernt programmeringsspråk som blivit populärt på grund av dess unika egenskaper som t.ex. minnessäkerhet utan att gå miste om prestanda. Inbyggda system är ett område där Rust har blivit mer vanligt under de senaste åren. Programmeringsspråket har även i flera år blivit valt till det mest populära språket bland dess utvecklare.

Det är finns inget direkt svar på om Rust är optimalt för användning i inbyggda system. Jämfört med de vanligaste programmeringsspråken för inbyggda system har Rust flera fördelar som t.ex. både minnessäkerhet och utmärkt dokumentation och Rust ligger inte långt efter om man jämför effektivitet. Minnessäkerheten uppnås genom ägarskapsmodellen som säkerställer att varje värde endast har en ägare vid en viss tidpunkt. Ägarskapsmodellen gör det möjligt för Rust att vara snabbare än andra moderna programmeringsspråk eftersom den inte kräver en sophanterare för att frigöra minne. Säkerheten gör det lättare att upptäcka buggar som buffertöverfyllningar och avreferering av nullpekare, vilka kan orsaka stora konsekvenser i kritiska system [9].

Eftersom Rust är ett relativt nytt programmeringsspråk saknas fortfarande funktionalitet inom mer specificerade områden. Det här begränsar möjligheten att tillämpa Rust på ett smidigt sätt i många inbyggda system. Förutsägbarheten för Rustkod i inbyggda system kan inte heller avgöras på grund av att endast få tester blivit utförda. Det är också värt att nämna att Rusts bibliotek sällan har uppnått stabila versioner. De här faktorerna begränsar i synnerhet användningen av Rust inom företag eftersom det ofta leder till mer arbete än förväntat. På grund av att Rust inte har uppnåt en bred utsträckning för inbygga system är också en stor del av biblioteken byggda som hobbyprojekt vilket inte garanterar underhåll eller kvalitet [3]. Att språket används som en hobby och inte för professionellt syfte kan vara en orsak till att Rust är det mest omtyckta programmeringsspråket.

Rust har böcker och kurser för att lära sig använda språket [8]. Programmerare anser ändå att Rust är ett programmeringsspråk som är svårt att lära sig, speciellt i början av lärandet. Orsaken till den svåra inlärningen beror på att Rust har stränga regler som inte är bekanta från tidigare programmeringsspråk. En stor mängd tillgängliga verktyg gör det också svårare att komma ihåg vilka funktioner som kan användas [3].

Användningen av Rust för Linuxkärnan kan leda till en större användning i framtiden. Utvecklingen av Linux för potentiellt med sig nya egenskaper och verktyg. Eftersom Linux är ett utvecklat operativsystem är chansen att nya Rustverktyg också skrivs och

underhålls bättre. Linux kan tillämpas i inbyggda system och möjligtvis gör det Rusts användning vanligare, åtminstone för Linuxbaserade lösningar [6].

Det ser ut som om programmeringsspråket Rust har en. Vilka områden Rust kommer vara starkast inom är oklart fortfarande oklart. För tillfället tillämpas Rust mest för säkerhetskritiska system som kräver hög prestanda. Tillämpningen i inbyggda system är svårare men i och med att Rust har grupper som jobbar specifikt för att göra språket bättre för inbyggda system finns en chans att språket blir mer användbart för detta ändamål.

6 Referenser

[1] D. N. Serpanos och A. G. Voyiatzis, ”Security Challenges in Embedded Systems,” *ACM Trans. Embed. Comput. Syst.*, årg. 12, nr 1s, 2013, ISSN: 1539-9087. URL: https://doi.org/10.1145/2435227.2435262.

[2] C. COSMIN, ”Rust – The Programming Language for Every Industry,” *ECONOMY INFORMATICS JOURNAL*, årg. 19, nr 1/2019, s. 45–51, sept. 2019. URL: https://doi.org/10.12948/ei2019.01.05.

[3] N. Borgsmüller, ”The Rust Programming Language for Embedded Software Development,” *Ph. D. Dissertation. Technische Hochschule*, 2021.

[4] D. Kleidermacher och M. Kleidermacher, *Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development*. Newnes, 2012.

[5] Z. Shao, C. Xue, Q. Zhuge, M. Qiu, B. Xiao och E.-M. Sha, ”Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software,” *IEEE Transactions on Computers*, årg. 55, nr 4, s. 443–453, 2006.

[6] S.-F. Chen och Y.-S. Wu, ”Linux Kernel Module Development with Rust,” i *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, 2022, s. 1–2.

[7] W. Bugden och A. Alahmar, ”Rust: The Programming Language for Safety and Performance,” 2022. URL: https://arxiv.org/abs/2206.05503.

[8] S. Klabnik och C. Nichols, *The Rust Programming Language*. No Starch Press, 2023.

[9] A. Pinho, L. Couto och J. Oliveira, ”Towards Rust for Critical Systems,” i *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, s. 19–24.

[10] T. Uzlu och E. Şaykol, ”On utilizing rust programming language for Internet of Things,” i *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*, 2017, s. 93–96.

[11] I. Culic, A. Vochescu och A. Radovici, ”A Low-Latency Optimization of a Rust-Based Secure Operating System for Embedded Devices,” *Sensors*, årg. 22, nr 22, 2022, ISSN: 1424-8220. URL: https://www.mdpi.com/1424-8220/22/22/8700.

[12] S. Georgiou, M. Kechagia, P. Louridas och D. Spinellis, ”What are Your Programming Language’s Energy-Delay Implications?” I *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, s. 303–313.

[13] R. Pereira m. fl., ”Ranking programming languages by energy efficiency,” *Science of Computer Programming*, årg. 205, s. 102 609, 2021, ISSN: 0167-6423. URL: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[14] I. Plauska, A. Liutkevičius och A. Janavičiūtė, ”Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller,” *Electronics*, årg. 12, nr 1, 2023, ISSN: 2079-9292. URL: https://www.mdpi.com/2079-9292/12/1/143.

[15] Noseda, Mario, Frei, Fabian, Rüst, Andreas och Künzli, Simon, ”Rust for secure IoT applications : why C is getting rusty,” en, 2022. URL: https://digitalcollection.zhaw.ch/handle/11475/25046.

[16] H. Ayers m. fl., ”Tighten Rust’s Belt: Shrinking Embedded Rust Binaries,” i *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2022, New York, NY, USA: Association for Computing Machinery, 2022, 121–132, ISBN: 9781450392662. URL: https://doi.org/10.1145/3519941.3535075.