T.C.
BEYKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ BİLİM DALI

RUST DİLİ İLE ÖNYÜKLEYİCİ TASARIMI

Yüksek Lisans Tezi

Tezi Hazırlayan: **Tunç UZLU**

İstanbul, 2016

T.C.
BEYKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ BİLİM DALI

RUST DİLİ İLE ÖNYÜKLEYİCİ TASARIMI

Yüksek Lisans Tezi

Tezi Hazırlayan:

Tunç UZLU

Öğrenci No:

140820004

Danışman:

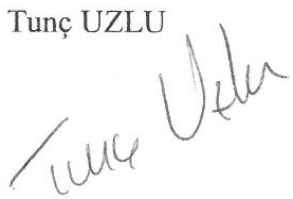
Yrd. Doç. Dr. Ediz Şaykol

İstanbul, 2016

YEMİN METNİ

Yüksek lisans tezi olarak sunduğum “Rust dili ile önyükleyici tasarımı” başlıklı bu çalışmanın, bilimsel ahlak ve geleneklere uygun bir şekilde tarafımdan yazıldığını, yararlandığım eserlerin tamamının kaynaklarda gösterildiği ve çalışmam içinde kullanıldıkları her yerde bunlara atıf yapıldığını belirtir ve bunu onurumla doğrularım. 01.12.2015

Tunç UZLU



T.C.
BEYKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

YÜKSEK LİSANS TEZ SAVUNMA SINAVI SONUÇ TUTANAĞI

Beykent Üniversitesi
Fen Bilimleri Enstitüsü Müdürlüğü'ne,

Aşağıda tez adı belirtilen yüksek lisans öğrencisi 140820004 no'lu TUNA UZLU 'in 16.5.2016 tarihinde yapılan tez savunma sınavı[1] sonucunda 45 dakika süreyle sunduğu ve savunduğu tezi hakkında[2] oybirliğiyle, KABUL kararı verilmiştir.

Bilgilerinize saygılarımızla arz ederiz.

Anabilim Dalı : Bilgisayar Mühendisliği
Programı : Bilgisayar Mühendisliği
Tez Başlığı[3] : Rust dili ile öngörleyici tasarımı

Tez Sınav Jürisi	Öğretim Üyesi	İmza
Danışman :	Yrd. Doç. Dr. Ediz SAYKOL	
Üye :	Yrd. Doç. Dr. Atılım Yılmaz	
Üye :	Doç. Dr. Gökhan SİLAHTAROĞLU	

---

[1] Jüri üyeleri söz konusu tezin kendilerine teslim edildiği tarihten itibaren en geç bir ay içinde toplanarak öğrenciyi tez savunma sınavına alır. Belirlenen günde yapılamayan jüri toplantısı, katılanların hazırladığı bir tutanakla enstitü yönetimine bildirilir. Bu durumda jüri en geç onbeş gün içinde toplanarak adayı tez savunma sınavına alır. Tez savunma sınav süresi en az 45 dakikadır. Yüksek lisans tez savunma sınavı, tez çalışmasının sunulması ve bunu izleyen soru-yanıt bölümlerinden oluşur ve dinleyiciye açıktır. (Beykent Lisansüstü eğitim ve Öğretim Yönetmeliği-Madde30-3)

[2] Tez sınavının tamamlanmasından sonra jüri, tez hakkında "kabul", "düzeltme" veya "red" kararı verir. Jüri başkanı, jüri üyelerince imzalanmış sınav tutanağını, tez sınavını izleyen üç gün içinde ilgili enstitü yönetimine teslim eder. Tezi başarısız bulunan öğrencinin Enstitü ile ilişiği kesilir. Tezi hakkında düzeltme kararı verilen öğrenci en geç üç ay içinde gerekli düzeltmeleri yaparak ve yönetmelikte belirtilen usullere uygun olarak tezini aynı jüri önünde yeniden savunur. Bu savunma sınavında da tezi kabul edilmeyen öğrencinin enstitü ile ilişiği kesilir. (Beykent Lisansüstü eğitim ve Öğretim Yönetmeliği-Madde30-4)

[3] İleride doğabilecek aksaklıkların engellenmesi için tezin başlığının yazılması gerekmektedir.

TEŞEKKÜR

Tez çalışmamda bana danışmanlık yapan ve çok değerli yönlendirmelerde bulunan Yrd. Doç. Dr. Ediz Şaykol hocama teşekkür ederim. Hayatimin her döneminde beni destekleyen aileme minnet ve şükranlarımı sunarım.

RUST DİLİ İLE ÖNYÜKLEYİCİ TASARIMI

Tezi Hazırlayan: Tunç Uzlu

Özet

Bu tez çalışmasında işletim sistemlerinin geçici hafızaya yüklenmesi ve başlatılması görevini üstlenen önyükleyicilerin Rust programlama dili ile X86 mimarisi için tasarımının aşamaları ve ortaya çıkabilecek problemler anlatılmaktadır. Önyükleyicilerin geçmiş sistemlerden bugünkü haline evrimleşmesine yer verilmekle birlikte yeni bir teknoloji olan Rust programlama dilinin sistem programlamaya getirileri ve işletim sistemi bileşenlerinin tasarımında nasıl rol oynayabileceği de gösterilmeye çalışılmıştır.

Anahtar kelimeler: Rust, önyükleyici, işletim sistemi

BOOTLOADER DESIGN WITH RUST LANGUAGE

Presented by: Tunç Uzlu

Abstract

In this thesis, we examine using Rust programming language for designing bootloaders which are responsible for loading and launching process of operating systems. We also mention evolution of bootloaders for X86 CPU architecture from older generation systems to today's modern systems. We are trying to emphasize that how Rust programming language performs for creating an operating system component and how it is changing systems programming.

Keywords: Rust, bootloader, operating system

İÇİNDEKİLER

Sayfa No.

TABLOLAR LİSTESİ

Sayfa No.

ŞEKİLLER LİSTESİ

Sayfa No.

KISALTMALAR

ABI	**:**	Application Binary Interface
ACPI	**:**	Advanced Configuration Power Interface
ADT	**:**	Algebraic Data Type
API	**:**	Application Programming Interface
ARC	**:**	Automatic Reference Counting
AST	**:**	Abstract Syntax Tree
BCD	**:**	Binary Coded Decimal
BIOS	**:**	Basic Input Output System
CD	**:**	Compact Disc
CWE	**:**	Common Weakness Enumeration
DMA	**:**	Direct Memory Access
DOM	**:**	Document Object Model
EEPROM	**:**	Electrically Erasable Programmable Read Only Memory
EFI	**:**	Extensible Firmware Interface
FFI	**:**	Foreign Function Interface
FPU	**:**	Floating Point Unit
GC	**:**	Garbage Collector
GCC	**:**	GNU Compiler Collection
GNU	**:**	GNU's Not Unix
GPT	**:**	GUIDed Partition Table
GUID	**:**	Globally Unique Identifier
ID	**:**	Identification
IO	**:**	Input Output
ISA	**:**	Industry Standard Architecture

JSON : Javascript Object Notation

LLVM : Low Level Virtual Machine

LPC : Low Pin Count

MBR : Master Boot Record

MMIO : Memory Mapped IO

MPS : Multi Processor Specification

NAND : Negative-AND

NOR : Negative-OR

PCI : Peripheral Component Interconnect

PE : Portable Executable (Windows)

RAII : Resource Acquisition Is Initialization

RFC : Request For Comments

ROM : Read Only Memory

RTTI : Runtime Type Information

SIMD : Single Instuction Multiple Data

SMBIOS : System Management BIOS

SMP : Symmetric Processing

SRAM : Static Random Access Memory

SSE : Streaming SIMD Extensions

UEFI : Unified Extensible Firmware Interface

USB : Universal Serial Bus

UTF : Unicode Transformation Format

X86 : Intel-Amd x86-64 işlemci mimarisi

1. GİRİŞ

Rust programlama dili Mozilla Foundation tarafından geliştirilen bir sistem programlama dilidir. Sistem programlama dilleri temel olarak C dili, nesne tabanlı ve meta programlama desteği bulunan devam niteliğindeki C++ dili ve makine dillerinin en yaygın olarak kullanıldığı bir gruptur. Bu diller genellikle işletim sistemleri ve aygıt sürücüleri gibi alt seviye donanım ile iletişim kurulan yazılımlarda ya da sinyal işleme, oyun motorları Web tarayıcıları gibi zamanlamanın kritik olduğu uygulamalarda kullanılır. Bu dillerde genellikle zaman içerisinde radikal değişiklikler olamamaktadır. Bunun en büyük sebepleri var olan eserlerin karmaşık yapısı ve uygulama geri uyumluluğu olarak düşünülebilir.

Örneğin C++ programlama dili içerisine atası olan C diline ait hemen hemen her özelliği taşır. Bunun getirisi var olan yazılımın bu devam niteliğindeki dile kolaylıkla taşınabilmesi veya öğrenim eğrisinin düşük tutulabilmesi olabilir. Fakat bu tasarım metodolojisi beraberinde inceleyeceğimiz birçok problemi de getirmektedir. Çünkü C dilini tamamen içine katmış olması bu dildeki tasarımsal problemleri veya hataya düşme noktalarında kapsaması anlamına gelir. İlk yıllarda C++ evrenine geçiş yapanlar için bu düşük öğrenme eğrisi oldukça fayda sağlamaktaydı. Fakat bu teknoloji ilerlemeye devam ettikçe geçmişten gelen bu yük ciddi tasarım ikilemlerine ve güvenlik kontrollerinin etrafından dolaşabilmeye zemin hazırlamaktadır.

Bu alanda yenilikçi bir tavır sergileyen fakat yayınlanmasından kısa bir sure sonra terkedilen bir başka dil ise Cyclone dilidir. Rust dilinin aksine sıfırdan yeni bir teknoloji olarak değil C dilinin üzerine bir eklenti olarak tasarlanmıştır ve dili daha güvenli bir hale getirmek için çabalamıştır.

“Cyclone eklentisinin ideali C dili ile tasarlanmış hali hazırda var olan kodların yoğun caba sarf etmeden taşınabilmesi olmuştur. Bu ideal en nihayetinde Cyclone dilinin semantik yapısını kısıtlamış ve Rust diline göre kullanması çok daha hantal bir sistem olmasıyla sonuçlanmıştır” [14].

Rust dili sadece geçmiş sistem programlama dillerindeki hataları çözmeye değil modern dillerde kullanılan ve kabul görmüş dil özelliklerine sahip olması acısından sınıfındaki örneklerinden farklıdır. Çünkü kendine has tasarlanmış semantikleri, bilgisayar bilimlerindeki C dili cağından bu yana gelişen hem fonksiyonel hem de **procedural** dillerdeki yenilikleri katmaya çalışması acısından sistem programlamaya yeni bir boyut kazandırmayı amaçlamaktadır.

Bu tez çalışmasının giriş bölümünde Rust programlama dili ve önyükleyicilere ait temel bilgiler verilmektedir. İkinci bolümde Rust programlama dilinin fonksiyonel elemanları örneklenmiştir. Üçüncü bolümde Rust diline ait gelişmiş semantikler incelenmiştir. Dördüncü bolümde Rust dilinin güvensiz yazılması anlatılmış, C dili ile arasındaki güvenlik modeli farklılıkları araştırılmış ve EFI önyükleyici aygıt yazılımı tanımlanmıştır. Beşinci ve son bolümde ise önyükleyici tasarımı için Rust dilinin konfigürasyonu tartışılmış ve kod örnekleri verilerek tez çalışması tamamlanmıştır.

1.1. Rust programlama dili

Rust programlama dili Graydon Hoare tarafından 2006 yılında geliştirilmeye başlanmıştır. Mozilla Foundation ilk olarak 2009 yılında projeyi fonlamaya başlamış ve 2010 yılında duyurmuştur. İlk stabil surum çıkana kadar dil birçok değişikliğe uğramıştır. Öyle ki stabil sürümden önceki kodları derlemek genellikle mümkün değildir. Örneğin birden fazla işaretçi türünün basitleştirilip bir tane türe indirgenmesi gibi temel değişiklikler çok sık yapılmıştır. Stabil surumun çıkmasıyla dil üretim seviyesinde de kullanılabilir hale gelmiştir [1].

Aynı zamanda Mozilla Foundation Web tarayıcısı motoru olan Servo[1] teknolojisini rust ile aktif olarak geliştirmektedir. Servo sayfa çiziminin paralel şekilde yapmayı hedefleyen şuan için geliştirilme aşamasındaki yeni bir teknolojidir.

"Servo tasarım itibari ile paralel olanakların mümkün oldukça fazla kullanılmasına dayanır. Servo motorunun benzersiz başarılarından bir tanesi **rendering** ve

[1] https://servo.org/

compositing aşamalarını aynı anda işleyebilmesidir. Geleneksel teknolojilerde bu asamalar art arda işlenir çünkü çalışmakta olan kod DOM ile dolaylı olarak etkileşime girmektedir" [18].

Rust ile birlikte popüler olan ve bir bakıma yarışan bir diğer dilde Go[2] dilidir. Google tarafından desteklenen Go, daha çok üst seviye programlama için uygun olarak tasarlanmıştır. Go hafıza modeli GC üzerine kuruludur ve jenerik türlere sahip değildir. Bu sayede öğrenme eğrisi oldukça düşük tutulmuş bir bakıma Python diline yakındır denilebilir. Rust hafıza güvenliği modelinden oldukça farklı olan Go, Rust dilinin aksine Thread çalışma ortamları arasında hafıza güvenliği garantisi sunmaz. Go dilinde concurrency için kullanılan Goroutine yapılarının arasında **data race** önlemek için bir mekanizması bulunmamaktadır [1].

Rust sistem programlama için son zamanların en çok konuşulan teknolojilerinden birisi olmuştur öyle ki en yaygın bulut dosya yedekleme şirketlerinden birisi olan Dropbox Petabyte boyutlarda depolama yapabilen Diskotech manikalarında Rust ile geliştirdikleri yazılımı kullanmaktadır [6].

Graydon Hoare bir röportajında Rust diline neden bu denli bir ihtiyaç olduğu sorusuna su şekilde yanıt vermiştir: "Başka dillerde bilinen, sevilen ve başarılı olduğu aşikâr fikirler yaygın kullanılan sistem dillerine entegre olmadı veya vasat hafıza modelleri bulunan (güvensiz ve **concurrency** düşmanı) dillere eklendiler. 70li yılların sonu ve 80li yılların başında bu dillerle rekabet edenlerden çok başarılı olanlar vardı ve ben bunların fikirlerini alıp hayata geçirmek istedim. Bu teoriden yola çıkarak koşulları değişti: internet yüksek oranda aynı anda çalışır durumda ve yüksek oranda güvenlik bilincine sahip. Dolaysı ile tasarım tercihleri sürekli olarak (örneğin) C ve C++ arasında gidip geliyor" [5].

Buradan yola çıkarak Rust dilinin temel felsefesi hafızayı hem paralel hem de tekil çalışan sistemlerde güvenli bir bicimde kullanmak ve bunu yaparken performans ile güvenlik arasında bir tercih yapmak zorunda kalmamaktır.

[2] https://golang.org/

Rust dili ve derleyicisi tamamen acık kaynak kodludur. Kaynak kodu Github[3] Repository üzerinden yayınlanmış durumdadır. Bu tezin yazıldığı zamanda 1336 kişinin projeye katkısı bulunmaktadır. Aynı zamanda dilin derleyicisi olan Rustc tekrardan Rust ile yazılmıştır. Bu yüzden yavaş derleme yaptığına dair eleştiriler almaktadır. Bu şekilde açık kaynak sistemin kendini projeye adamış tasarımcılar ile Mozilla Foundation bünyesine çalışanların yani sıra Rust kullanıcılarının da projeye katkıda bulunması mümkün kılınmaktadır. Dilin standartları sistemin mimarları tarafından kararlaştırılır. Standardize edilmiş modüller Github Repository üzerinden RFC olarak yayınlanır[4] ve Issue sistemi üzerinden tartışmaya açıktırlar. Böylelikle tasarıma dair kararların nasıl bir yol izlendiğini takip etmek ve dilin arka planı hakkında bilgi sahibi olmak mümkündür. Rust böylelikle kullanıcının sistemine herhangi bir yazılım gibi kurulabilir. Şuan için Windows, Linux ve OS X işletim sistemleri için hazır derlenmiş yükleyicileri bulunmakla birlikte Linux ve Macintosh işletim sistemi üzerinde Autotools ve Make araçları ile Ios, Android, Rasperry Pi ve daha birçok işletim sistemini hedef alarak derlenebilmektedir. Böylelikle rust C dilinin geliştirme ortamına göre işletim sisteminden çok daha soyut yani **decoupled** yapıdadır. Derleyici veya standart kütüphane gibi C ve türevi dillere ait geliştirme araçları işletim sistemi veya çekirdek API içerisinde bulunurlaralar. Bu durumda Rust bu soyut tasarımı sayesinde **deterministic** sonuç üretmeye çok daha yakındır. Çünkü işletim sistemi konfigürasyonu veya çekirdek API ile doğrudan bağlantılı değildir.

"Rust sürümleri Beta, Nightly ve Stable olmak üzere üç kanaldan yayınlanır. Nightly güncellemeler günlük olarak mevcuttur. Her altı haftada bir en son Nightly surum Beta olarak yükseltilir. Bu noktada sadece ciddi hataların düzeltilmesi yapılacaktır. Altı hafta sonra bu Beta surum Stable olarak yükseltilir ve bu yeni 1.x surumu olacaktır. Stable kanalı genel kullanıcı kitlesi için en uygun olandır ve ciddi bir ihtiyaç bulunmadıkça bu kanalın kullanılması önerilir" [17].

İşletim sistemleri ve aygıt sürücüleri mutlak suretle ve gömülü sistem yazılımları genellikle rust Nightly sürümüne ihtiyaç duyarlar. Bunun en büyük sebebi dile eklenen deneysel özellikleri kullanmak, **inline** şekilde Assembly yani makina dili kodu eklemek ve standart kütüphane üzerinde değişiklik yapmak ve hatta çalışmanın

[3] https://github.com/rust-lang/rust
[4] https://github.com/rust-lang/rfcs

devamında kullanılacağı gibi standart kütüphaneyi tamamen kaldırabilmektir. Rust Stable surumu ile derlenecek bir işletim sistemi bileşeninin doğası gereği tüm hafıza modelini kullanabilmesi beklenemez. Bu durumdan yola çıkarak bazı alt seviye Rust özellikleri sadece Nightly sürümünde bulunmaktadır. Fakat Rust dili sunduğu hafıza güvenlik modelinin büyük bir kısmını çalışma esnasında değil derleme esnasında yaptığı kontroller ile sağlar. Doğası gereği çalışma zamanında kontrole ihtiyaç duyan elemanları, örneğin Array veya Any yapıları gibi, açıkça belirlenmiştir. GC yapısına ihtiyaç duymadan güvenli bir hafıza modeli sunan Rust bu şekilde standart kütüphane olmasa dahi Libcore adi verilen basit ve işletim sisteminden bağımsız çekirdek kütüphanenin de yardımı ile birlikte bu hafıza güvenli modeli işlemeye devam eder.

1.2. Gömülü sistemler

Sistem programlama dilleri gömülü sistem yazılımları için de temel kabul edilmektedir. Gömülü sistemler özel bir bilgisayar çeşididir ve bu bilgisayar kontrol ettiği aygıtın içinde gömülü durumdadır. Düşük hafıza veya işlem gücü bu donanımlar uygulama tasarımcılarını çalışma zamanı açısından performans maliyeti olmayan teknolojilere yönlendirir. Bu bilgisayarlar sıradan sunucu, masaüstü ve dizüstü bilgisayarlardan çok daha küçük olmakla birlikte sayıca çok daha çeşitlidir [13].

Sensör gibi çok basit gömülü sistemlerde Moore’s Law faydaları yeni teknolojinin boyut ve maliyet olarak azalması için kullanılacak fakat işlem kapasitesi artmayacaktır [19]. Sistem programlama dillerinin gömülü sistemler için vazgeçilmez olmasının temel sebebi hafızayı verimli kullanıyor olmalarıdır. Çünkü bu dillerde hafıza yönetimi otomatik olarak dile ait bileşenler tarafından yapılmaz. Fakat üst seviye diller çoğunlukla GC birimi ile hafıza yönetimi yaparlar. GC hafıza birimlerinin oluşumunu takip etmek amacıyla işaretler. Bu birimler kullanım dişi kaldıklarında bunları silerek tekrar boş hafıza arasına katar. Fakat burada iki temel problem vardır. Bir tanesi silme işleminin zamanlamasının tamamen GC biriminin elinde olması. Bu kısıtlı işlem gücü olan gömülü sistemin zaman acısından kritik bir işlem yaptığı ana denk geldiğinde performans düşüşüne yol acar. Bir diğer ve daha önemli bir sorun ise silinmesi gereken hafıza bloklarının anında silinip boş hafıza hızlı bir şekilde kazandırılamamasıdır. Zaten hafızası oldukça kısıtlı olan gömülü sistemlerde bu ciddi bir problemdir.

GC **deterministic** olmayan aksamalar ortaya çıkartır [16]. Bununla birlikte GC ciddi bir hafıza güvenliği sağlamaktadır çünkü hafıza otomatik olarak dile ait bir birim tarafından yönetildiğinden hafıza bloğunun yanlışlıkla iki defa silinmesi veya silinmesinin unutulması gibi insan kaynaklı hataların olmasını engellemektedir. Rust dilinin oldukça önem taşıyan sahiplik sistemine bu konuda yenilik getirmektedir. Bu sistem GC benzeri bir güvenlik seviyesini derleme esnasında sunmakta dolaysı ile çalışma zamanında hiçbir performans kaybı yaşanmamaktadır. Rust bize GC ile çalışan üst seviye dillerin güvenliğini C gibi hafif fakat güvensiz bir dilin hızında sağlamaktadır.

"Kaynak açısından kısıtlı mikroişlemciler enerji tüketimi, hafıza boyutu ve çalışma zamanı acısından zorluklarla karşılaşarak işletim sistemi tasarımını benzeri görülmemiş bir cirimde etkilemektedirler. Özellikle gömülü işletim sistemleri çok daha güvenilebilir olmalı ve genel amaçlı işletim sistemlerinden çok daha az hafıza kullanmalıdırlar" [16].

C++ en kudretli sistem programlama dili olmasına rağmen genellikle gömülü sistemlerde C dilinin gerisinde kalır. Öyle ki C++ için birçok yönden sistem programlama dillerinin 8000-pound ağırlığındaki gorili benzetmesi yapılmıştır [14]. Örneğin C++ dilinin RTTI gibi hantal ve karmaşık özellikleri de yoğun işletim sistemi desteği olmadan anlamsız kalır ve genellikle gömülü sistemlerde tercih edilmezler.

Derleyiciler daha optimize kod yaratmak amacıyla Instruction komutlarının yerlerini değiştirebilir ve hatta sonucu etkilemediğini fark edip tamamen silebilirler. Bu yer değiştirme hem okuma hem de yazma komutları için yapılabilir. İşletim sistemlerinde derleyiciye anlamsız gelebilen ve optimizasyon aşamasında müdahale etmek isteyeceği komutlar olacaktır. Bunlar bir donanımın hazır duruma gelmesi için beklemek, sonucun hiç kullanılmadığı hafıza adreslerine yazmak veya okumak olabilir. C dillerinde bu tip işlemler için kullan değişkenler Volatile anahtar kelimesi ile tanıtılır. Böylelikle derleyici bu değişkenlere bağlı işlemler üzerinde optimizasyon uygulamaz. Rust dilinde bu tip derleyici müdahalelerine Intristic adi verilir ve bunlar dilin içine gömülü değil standart kütüphanenin bir parçasıdır. Örneğin 'std:: intristics:: volatile_load' fonksiyonu geçici hafızadan Register ünitesine Volatile yükleme yapar. C ve C++ dillerinin aksine okuma ve yazma ayrı ayrı kullanılabilir. Bu 'std:: intristics'

modülünde direkt olarak işlemci Instruction komutlarına karşılık gelen fonksiyonlar bulunmaktadır. Dolaysı ile Rust dilinde Intristic fonksiyonlar değişken her okunduğunda veya yazıldığında kullanılmalıdır. Bu tasarımcıya daha detaylı bir kullanım alanı sunar çünkü bir değişken örneğin yazım için Volatile olabilir ve fakat okuma için Volatile olmayabilir [13].

1.3. Temel önyükleyici kavramları

Önyükleyiciler işletim sistemlerinin ilk basamağıdır. Bir önyükleyici görevine işletim sistemini bulunduğu ortamdan geçici hafızaya kopyalamakla baslar. Bu ortam kişisel bilgisayarlar için sabit disk veya katı hal diski, gömülü sistemler için NAND veya NOR gibi teknolojilerle üretilmiş katı hal kalıcı hafıza birimidir. İşletim sistemi kurulumu esnasında bu ortam CD veya günümüzde daha karmaşık olan USB arabirimi ile bağlı bir depolama ünitesi veya ağ üzerinde olabilir.

Önyükleyicinin sonraki görevi işletim sisteminin kendini başlatması, sistemin konfigürasyonu ve donanımı hakkında bilgi edineceği aygıt yazılımı tarafından doldurulan tabloları veya geçici hafızadaki adresini sağlamak olacaktır. X86 mimarisinde bunlar ACPI, SMP ve SMBIOS gibi donanım bilgisinin bulunduğu ve aygıt yazılımı aracılığı ile donanıma hükmetme olanağı tanıyan alt seviye veri yapılarıdır. X86 mimarisinde yakın zamana kadar bu tabloların yaratılması ve önyükleyicinin başlatılması gibi donanım etkileşimi BIOS adi verilen yazılım tarafından yapılırdı. BIOS bu mimaride genellikle ana kart üzerindeki ROM içerisinde bulunan, zorunlu durumlar dışında güncellenmeyen oldukça kapsamlı
ilk çalışan yazılımdı.

Aygıt sürücülerinin kullanılmadığı eski işletim sistemleri tüm donanım erişimi BIOS fonksiyonları üzerinden yapılırdı. Zamanalar bu yöntemin performans ve güvenlik acısından yetersiz kalmaya başladığında işletim sistemleri her aygıt için farklı aygıt sürücüsü kullanmaya başladı. Bu sürücüleri çekirdek seviyesi adi verilen ve donanıma direkt erişimi bulunan en alt halkada konumlandırdılar. Öyle ki daha yüksek halkalarda çalışan kullanıcı yazılımları veya işletim sistemine ait fakat çekirdek görevi yapmayan ara yazılımlar donanıma erişemeyecekti. Bu şekilde

performans ve güvenlik büyük ölçüde artmış oldu. Çünkü tarihsel sistemlerde her uygulama BIOS fonksiyonlarına erişiyordu. Uygulamaların birbirinden ve hatta işletim sisteminden bağımsız hafızası yoktu.

BIOS sisteminin asil amacı aygıt sürücülerinin olmadığı zamanlarda donanım etkileşimi kurmaktı. Aygıt sürücülerin kullanımının yaygınlaşmasıyla bu fonksiyonlar sadece önyükleyiciler tarafından kullanılmaya devam etti. Fakat BIOS mimarisi onlarca yıl organik şekilde geliştirildi ve modern bilgisayarın ihtiyaçlarına cevap veremez duruma geldi. X86 işlemci mimarisi beklide bilgisayar bilimleri tarihindeki en başarılı geri uyumluluk örneklerinden birisini sergilemektedir. Fakat bu geri uyum önyükleyici tasarımını oldukça karmaşık hale getirmektedir.

Bir X86 mimarisine sahip işlemci başlatıldığında Real-mode adi verilen 16-bit Register boyutu ve hafıza yolu destekleyen ve hafızanın sadece 1MB kısmına erişebilen kip ile çalışmaya baslar. Modern ve komplike olanalar dahil tüm işletim sistemleri ve önyükleyiciler bu sistem ile başlayıp ileri teknolojileri başlatmalıdır. Bu her ne kadar işletim sisteminin görevi olsa da önyükleyiciler işletim sisteminin çalışabileceği asgari ortamı hazır etmekle görevlidir. Örneğin 32-bit bir işletim sistemi bu boyutta veri yolu ve Register ünitesinin varlığını şart koşabilir. Bu durumda önyükleyici Protected-mode adi verilen moda geçmek ve ardından işletim sistemini başlatmak durumundadır. Bazı işletim sistemleri ISA veri yolu kontrolcüsü gibi birimlerin temel bir kullanılabilir bir durumda olmasını bekleyebilir bekleyebilir. Çekirdek ve işletim sisteminin aygıt sürücüsü birimi her ne kadar çalışmaya başladıktan sonra tüm donanıma tekrardan konfigürasyon yapacak olsa da bazı donanım kontrolcülerine veya Instruction komutlarına erişim sadece Real-mode kipi üzerinde var olmasıdır. Çekirdek tekrardan Real-mode kipine geçiş yapmayı istemeyebilir veya bu mimari gereği mümkün olmayabilir. Unutulmamalıdır ki X86 mimarisi standart bir Instruction kümesine sahip olsa da farklı donanım üreticileri tarafından üretilen birçok işlemci bu mimariyi kullanmaktadır. Hafıza kontrolcüsü veya LPC gibi tarihi donanımlar hala kullanılmadır ve bazı durumlarda donanıma bir kez konfigürasyon uygulanma imkânı bulunmaktadır. Tabii ki bunlar önyükleyiciler acısından uç senaryolardır. İşletim sistemi çekirdeği kipler arası stabil geçişler yapabilir ve hatta tarihi işletim sistemlerinde eski işlemciler için tasarlanmış uygulamaları çalıştırmak için performans kaybı göz önünde bulundurularak rutin

olarak Real-mode geçişi kullanılmıştır. Bu geçiş için Virtual-8086 kipi adi verilen donanım desteği de bulunmaktadır [20].

Protected-mode 32-bit hesaplamanın yapılabileceği kiptir. Burada Paging yapılabileceğinden ismini bu şekilde almıştır. Çünkü Paging ile hafıza sayfalara bölünebilir ve sanal hafıza kullanılabilir. Ayrıca Paging ile her sayfanın bağımsız okuma, yazma veya çalıştırma gibi güvenlik etiketleri bulunabilir. Bu sayede örneğin kod için ayrılan sayfa çalıştırılabilir fakat üzerine yazılarak değiştirilemez. Ayrıca çekirdek sayfalarına kullanıcı uygulamalarına ait halkadan çalıştırılan koddan erişilemez. Paging sayesinde sanal adresleme yaparak hafızayı kalıcı disk üzerinde takas alanı oluşturarak arttırmak mümkündür. Aynı zamanda yığın boyutunu dinamik olarak arttırmak için Guard Paging kullanılabilir. Uygulama yığını üzerindeki sayfaya yazma izni verilmez. Uygulama buraya yazmak istediğinde Page Fault (X86 mimarisinde çoğu hata kısa bir kod ada sahiptir. Örneğin #PF hata kodu Page Fault için kullanılır) hatasına yol acar. İşlemci hataya ilişkin adresleri Register belleğine koyar ve bir Interrupt oluşturur. Çekirdek uzayında çalışma durur ve bahsi gecen adrese bir sayfa tahsis edilir ve uygulama bir önceki Instruction ile tekrardan çalıştırılırdı. Instruction Pointer adresinden (X86 mimarisinde kısaca 32-bit ise EIP veya 64-bit ise RIP adi verilmiştir) çıkartma işlemi yapılarak bu lokaysan kolaylıkla hesaplanabilir. Bu şekilde uygulama farkında olmadan daha büyük bir yığın ile çalışmaya devam eder. Tabii ki işletim sistemi Linker çıktısını da göz önüne alarak yığın için azami bir boyut belirler ve bu noktaya gelindiğinde uygulama yığın taşması sebebiyle durdurulur. Long mode ise 64-bit ve güncel olarak kullanılan kiptir. Burada Protected-mode ile gelen tüm güvenlik mekanizmalarına ek olarak Canonical formda hafıza adresleme kullanılır. Burada tarihi Virtual-8086 veya BCD Instruction komutlarını kullanmak artık mümkün değildir. Paging ise Protected-mode ve Long-mode kiplerinde zorunludur. Tarihi önyükleyiciler Real-mode kısıtlamaları sebebiyle iki etap çalışmaktaydılar. İlk etabın amacı daha yüksek hafıza yükselmek, daha güvenli bir geçmek ve asgari Interrupt tablolarını oluşturmaktı. İkinci etap ise işletim sistemini yükler, yüksek hafızali kiplerin gereksinimine göre hafızayı sayfalara böler, yeni Interrupt tablosunu oluşturur ve işletim sistemini çalıştırır. Günümüzde ise Intel tarafından yayınlanan ve şuan birçok kurumun katkısı ile geliştirilmeye devam edilen EFI platform aygıt yazılımı kullanılmaktadır.

2. STATİK DİLLER VE FONKSİYONELLİK

Yazılım mimarları genellikle geliştirdikleri projelerde hata yapacaklarını kabul ederek yola çıkmazlar. Oysa ki analizden geliştirmeye testten uygulamaya birçok alanda oluşan sorunlar insan hatalarından meydana gelir. Yapılan hataları çözmek için kimi zaman projeye bastan başlanır veya her şey sıfırdan planlanır. Bazen olduğu yerden hataları çözmek ve her şeye kalınan yerden devam etmek mümkündür. Ama en kotu durumda mutlak başarısızlık tarih boyunca sayısız kere görülmüştür.

"Bu zararların maliyeti logaritmiktir – öyle ki, zamana göre on kat artmaktadır. Bir hata teknik şartname aşamasının başlarında bulunup düzeltildiğinde bunun sıfır ile örneğin 10 Cent aralığında bir maliyeti olur. Eğer aynı hata yazılım kodlanana ve test edilene kadar bulunamazsa bunun maliyeti 1 dolardan 10 dolara kadar çıkar. Eğer bu hatayı kullanıcı bulursa bu miktar kolaylıkla 100 dolar seviyesine çıkmaktadır" [21].

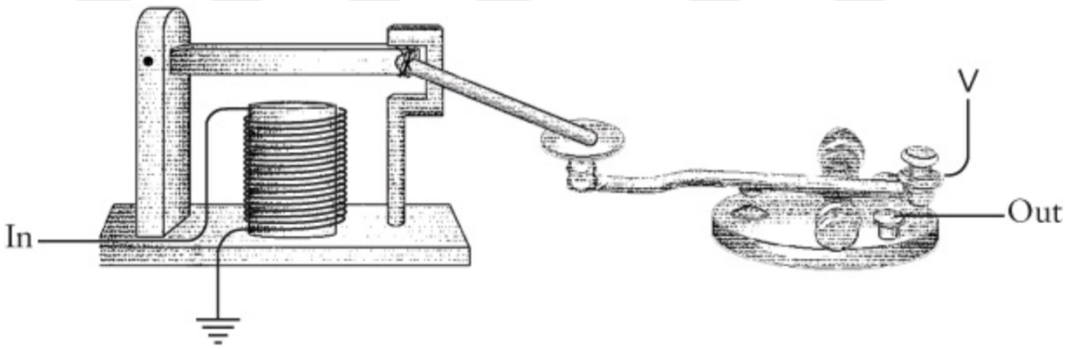
2.1. Statik türler

Statik türler bir programlama dilinin elemanlarına soyut türler atamaktır. Bu türlerin tüm sistem içerisindeki tutarlı kullanımını kontrol ettiğimizde bu soyut etiketleme birçok insan hatasını yakalamaya yardımcı olur. Bu türlerin kullanıldığı diller statik diller olarak anılır. Sistem programlama dillerinden C ve C++ ve tabii ki Rust statik dillerdir. Bu tür doğrulaması derleme aşamasında yapıldığından sistemince yakalanan hatalar makina kodu oluşturulmadan yakalanır.

Dinamik diller ise tür kontrolünün olmadığı dillerdir. Bunlara örnek olarak Python, Javascript gibi diller verilebilir. Bu sınıftaki diller genellikle derleyici tarafından derlenmezler. Derlendikleri durumlarda ise tür kontrolünden geçmezler. Ayrıca makina kodu yerine farklı bir dile okunaklı şekilde derlenebilirler. Bu dillerdeki amaç öğrenme eğrisini en alt seviyede tutmak ve geliştirme aşamalarını azaltmaktır. Düşük öğrenme eğrisi çok kısa surede bu dilleri kullanarak sonuç almaya olanak sağlar. Fakat bu düşük yatırım zamanla daha büyük problemlerin ortaya çıkmasına sebep olur. Bu sorunlarla karşılaşmak projenin kapsamının genişlemesi veya proje üzerinde çalışan insan sayısının artmasına bağlıymış gibi gözükse de aslında en küçük sistemde dahi

tür sisteminin yokluğu ilk andan itibaren birçok insan hatasının gözden kaçmasına sebebiyet verir. Çünkü tür sistemi güçlü ise hata yakalama potansiyeli de fazladır. Rust hem statik hem de güçlü bir tür sistemine sahiptir. Statik tür sistemine sahip fakat zayıf statik türler kullanan diller de mevcuttur. Bu dillerde türler arası otomatik cevrim söz konusudur. Dolayışı ile güçlü statik dillere göre tür sistemi açısından oldukça tutarsızdırlar. Dinamik diller tür kontrolüne sahip olmamalarından ötürü derleyiciye ihtiyaç duymamaları oldukça hafif ve hızlı bir geliştirme ortamı sağlar. Fakat bu insan hatalarına karşı büyük bir açık oluşturur. Derleyiciler hemen hemen her zaman hata bulmakta insanlardan çok daha beceriklidir.

Derleyicilerin insanlara kıyasla hata yakalama basarisini daha iyi tarif edebilmek için telgraf sisteminden bir örnek kullanılabilir. Charles Petzold bir kitabında telgraf tekrarlayıcı veya rölesi olarak bilinen cihazı hayali bir telgraf röle operatörünün ses cihazının demir çubuğu alçaldığı anda anahtara basıp yükseldiği anda çekmesinden yola çıkarak anahtar ve ses cihazı çubuğunu dışarıdan aldığı bir bir tahtayla bağlaması ile örneklemiştir [3].



Şekil.1. Temsili bir telgraf rölesi

Kaynak: Petzold, C. (2009). Code: The Hidden Language of Computer Hardware and Software içinde. [3]

Tıpkı duyduğu her Morse kodunu anahtara basarak tekrarlayan bir operatörün hata yapması olasılığının şekildeki temsili bir röleye göre çok daha fazla olacağı gibi dinamik bir dilde tür sisteminin yokluğundan kaynaklı insan hatalarının oluşma olasılığı da oldukça fazla olmaktadır.

Rust LLVM derleyici alt yapısını kullanır. Bu modern C ve C++ geliştirme ortamları tarafından da kullanılmaktadır ve dolayışı ile Rust uygulamaları C++ ve diğer sistem programlama dillerine ait optimizasyonlardan faydalanabilir. Rust uygulamaları tıpkı C dilleri gibi LLVM ara formuna derlendikten sonra bu diller ile aynı arka uç ile makina koduna çevrilmektedirler [1]. Hata ayıklayıcı olarak yine LLVM alt yapısı kullanılmaktadır. Rust-lldb adli modül sayesinde hata ayıklayıcı çıktıları daha okunaklı olacak şekilde ayarlanabilmektedir. Rust derleyici esnasında sunduğu hafıza güvenlik modeli sayesinde hata ayıklayıcıya düsen görevi hafifletmektedir. Geleneksel sistem programlama dillerinde hata ayıklayıcılar güvensiz hafıza bozulmaları sebebiyle geliştirme aşamasında rutin olarak kullanılan bir araç haline gelmiş durumdadır.

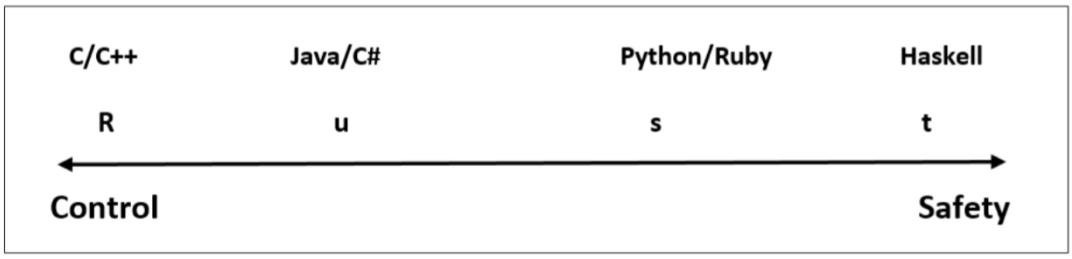
Derleyiciler aynı zamanda optimizasyon kabiliyetleri de insanlara göre çok daha iyidir. Dinamik dillerde projenin kararlılığını düşürmek pahasına yapılan birçok optimizasyon derleyici tarafından hiçbir risk almadan yapılabilmektedir. Gerektiğinde derleyici optimizasyonunu kapatmak veya düşük seviyelerde tutmakta mümkündür çünkü kaynak üzerinde bir değişiklik yapmayı gerektirmez.

2.2. Rust dilinin fonksiyonel elemanları

Fonksiyonel paradigma imperatif paradigmanın tersine bir veri kümesi ve buna uygulanan fonksiyonlar seklinde bir yapıdır. Genellikle matematiksen gösterime diğer programlama paradigmalarına göre çok daha fazla benzer. Fonksiyonel yapının daha iyi bir sistem olduğu aşikârdır. Genellikle fonksiyonel dillerin imperatif türevlerine tercih edilmesinin sebepleri öğrenme eğrisinin yüksek oluşu veya kaynakların yetersiz olması olarak düşünülse de bu sebep her zaman teknik olmayabilir.

Fonksiyonel diller sistem programlamada tercih sebebi değildir. Çünkü bu dillerin çoğunda donanım ile direkt etkileşime geçmek zaten mümkün değildir. Tamamen saf fonksiyonel dillerde işletim sistemi bileşenleri gibi imperatif mantıkla çalışan ve sadece IO etkileşimi yapan tasarımlar yapılması dili **idiomatic** olmayan şekilde kullanmaya sevk edecek ve amacından saptıracaktır. Bu problemin temel sebebi hafızaya direkt olarak müdahale edilememesidir. Üst seviye dillerin hemen hemen

hepsinde bu kısıtlama mevcuttur. Genellikle GC yapısı ile hafıza derleyici ve dilin çalışma zamanı eklentileri ile otomatik olarak idare edilir. Çok güçlü bir tür sistemiyle birleştiğinde bu sistem yüksek kararlılık ve düşük hata potansiyeli sunmaktadır. Bir diğer sebep ise fonksiyonel dillerim tembel hesaplanır oluşudur. Tembel hesaplanan dillerde hesaplamalar gerçekten sonuca ihtiyaç duyulduğu ana kadar ertelenirler. Bu donanımla zaman acısından kritik bir şekilde işlem yapmak için oldukça verimsiz bir sistemdir. Ayrıca fonksiyonel diller genellikle makine koduna değil yorumlayıcı üzerinde çalışmak için ara bir forma derlenirler. Direkt olarak makina diline derlenenler ise oldukça büyük ve karmaşık bir çalışma zamanı kütüphanesine sahiptirler ve de bu birim olmadan kullanılmaları oldukça zordur.



Şekil.2. Çeşitli dillere ait kontrol-güvenlik grafiği

Kaynak: Balbaert, I. (2015). Rust Essentials içinde. [1]

Rust alt seviye bir dildir. Hafızaya erişim açısından geleneksel alt seviye sistem dillerinden hiçbir eksiği yoktur. Rust hem imperatif hem de fonksiyonel paradigmayı benimser. Bu çoklu paradigması sayesinde saf fonksiyonel bir dil değildir ve imperatif paradigma ile geliştirme yapılabilir. Standart kütüphane dahilindeki fonksiyonel veri yapılarının bazıları tembel şekilde çalışmakta fakat dilin kendisi tembel değil **strict** şekilde hesaplanmaktadır.

"Neden fonksiyonel veri yapıları imperatif karşılıklarından daha zor tasarlanır ve uygulamaya konulurlar? İki basit problem bulunmaktadır. İlk olarak verimli veri yapıları tasarlamak ve uygulamaya koymak açısından bakıldığında fonksiyonel programlamanın yok edici güncellemelere (eşitlemelere) karşılık kısıtlayıcı olması sarsıcı bir handikap, bir usta şefin bıçağını almaya eşittir. Tıpkı bıçaklar gibi yok edici eşitlemeler yanlış kullanıldıklarında tehlikeli olabilirler fakat doğru kullanıldıklarında olağanüstü şekilde ekilidirler. İmperatif veri yapıları çoğunlukla eşitlemelere hayati

şekilde bağlı durumdadır ve fonksiyonel programlar için çok farklı çözümler bulunmalıdır" [4].

Örneğin Basel problemindeki sonsuz seriyi ilk 20 adim kullanılarak Rust dili ile fonksiyonel olarak hesaplanmak istendiğinde öncelikle sonsuz tamsayılar içeren bir Iterator kümesi ile başlanır. Bu kümedeki her sayının karesi hesaplanır, bu sonuç ondalıklı sayılara çevrilir ve tersi hesaplanır. Rust RFC212 standardı aksi belirtilmediği surece ondalıklı sayı türünün 64-bit yani 'f64' türü olduğunu yazmaktadır. Günümüz X86 işlemcileri ondalıklı sayı hesaplamaları için 80-bit 8087 yan işlemcisi yerine SSE kullandığından bu oldukça uygun bir secim gibi gözükmektedir. Bu hesaplama için Map yüksek mertebe fonksiyonu kullanılarak kümedeki her eleman üzerinde anonim bir fonksiyon çalıştırılır. Map bize yeni bir küme dönmektedir ve bu fonksiyon Rust dilinde yeni bir küme elde etmek için kullanılır. Yan etkiler için kullanımı **idiomatic** değildir. Daha sonra bu kümenin ilk 20 elemanı alınır. Bunun için Take fonksiyonu kullanılır ve bu fonksiyon ile yeniden yeni bir küme elde edilmiş olur. Rust dilindeki Iterator yapısı tembel olduğundan bu ana kadar herhangi bir hesaplama yapılmamış durumdadır. İmperatif bir dilde sonsuz bir hesaplama yapılmaya başlanmış olacaktı. Son olarak küme üzerine Fold fonksiyonu uygulanır. Fold diğer hesaplama fonksiyonlarının aksime tüketici bir fonksiyondur ve mutlak bir sonuç elde etmemize olanak tanır.

```
1  let basel_20 =
2      (1..)
3      .map(|n| { 1.0/(n*n) as f64 })
4      .take(20)
5      .fold(0.0, |acc, n| acc + n);
```

Diğer tüketici fonksiyonlar arasında kümedeki elemanların toplamını elde etmek için Sum fonksiyonu veya küme elemanlarını örneğin Vec gibi konteyner yapısına aktarmak kullanılan Collect fonksiyonu sayılabilir. Fold tıpkı Map gibi kümedeki tüm elemanlar üzerinde çalışır fakat Map fonksiyonundan farklı olarak bir başlangıç ve akümülatör değeri alır. Her tekrarlamada çalıştırılan anonim fonksiyon sonucu akümülatör değişkeni içeresine günceller.

3. TEMEL RUST KAVRAMLARI

Rust paketi Rustc derleyicisinin yani sıra gibi birçok yardımcı araç ile birlikte gelir. Örneğin Rustdoc otomatik olarak dokuman yaratmaya olanak sunar. Üçüncü parti Rust paketleri crates.io[5] internet sitesi üzerinden indirilebilir. Cargo paket yöneticisi de bu site üzerinden indirme yapmaktadır. Kütüphanelerin yani sıra geliştirme araçları da Cargo tarafından derlenebilir ve uygulama olarak kullanılabilir. Örneğin Rustfmt kodun şeklen Rust formatına uygunluğunu kontrol eder ve yer değiştirmeler yaparak uygun forma getirir [25]. Racer ise üçüncü parti bir geliştirme aracıdır. Metnin verilen noktasından uygun devam önerileri sunarak kelimelerin kalan kısmini otomatik olarak tamamlamaya yardımcı olmaktadır.

3.1. Cargo paket yöneticisi

Rust ekosistemini diğer sistem programlama dillerinden ayıran en önemli farklardan birisi şüphesiz ki Cargo isimli paket yöneticisidir. Cargo ile derlenebilen Rust paketlerine Crate adi verilir. Bir Crate en az bir ‘rs’ kod dosyası ve Toml formatında Cargo.toml isimli meta bilgi dosyasından oluşur. Kod dosyasının adi kütüphane türündeki Cargo projeleri için ‘lib.rs’ uygulama türündekiler için ise ‘main.rs’ olarak (Cargo kısa adları ile Lib ve Bin) adlandırılmaktadırlar. Kütüphane türündeki boş bir Cargo projesine ait örnek Toml dosyası aşağıda verilmektedir.

```
1  [package]
2  name = "dummy_crate"
3  version = "0.0.1"
4  authors = ["Tunc Uzlu <tunc@macbook.local>"]
5  [lib]
6  name = "library_1"
7  crate-type = [“rlib”, “dylib”]
```

[5] https://crates.io

Cargo projeye ait birçok işlemin merkezîleştirilmiş şekilde çalıştırılmasına olanak sağlar. Projeye ait iç içe klasörlerden herhangi birinden çağrılması durumunda kok klasörü otomatik olarak bulma özelliğine sahiptir. Build argümanı ile derleme, Doc ile otomatik olarak dokuman oluşturma, Test komutu ile test olarak etkiletenmiş bölümleri çalıştırma özelliği vardır. Cargo paket yöneticisi projenin bağımlı olduğu Rust paketlerini indirir, derler ve kullanıma hazır hale getirir.

Cargo proje bağımlılıklarını otomatik olarak indirme ve derleme özelliğine sahiptir. Her bağımlılık için net surum bilgisi tanımlamak mümkündür. Cargo.toml Dependencies bolumu eklenir. Burada gösterildiği gibi Log kütüphanesinin sabit olarak '0.2.5' surumu kullanılmaktadır. Aynı şekilde '0.2.*' seklinde sadece minör değişiklikleri almak mümkündür. Mac kütüphanesinin ise en yeni surumu kullanılmaktadır [1].

```
1 [dependencies]
2   log = "0.2.5"
3   mac = "*"
```

Cargo '–target' argümanı ile kullanılarak proje derlemenin yapıldığı geliştirme sisteminden farklı bir mimari hedefe derlenmesi sağlanabilir [13]. Bu şekilde farklı platformlara çapraz derleme yapmak mümkündür.

Benzer bir şekilde '--features' argümanı ile durumsal derleme mümkün kılınır. Örneğin aynı görevi üstlenen A ve B kütüphanelerine ait modülleri durumsal olarak eklemek mümkündür.

```
1 #[cfg(feature = "lib-A")]
2 pub mod lib_a;
3 #[cfg(feature = "lib-B")]
4 pub mod lib_b;
```

Örneğin A kütüphanesi ile derleme yapılacağı zaman yukarıdaki örneğe göre Cargo build '–features lib-A' seklinde çağrılmalıdır. Trait yapılarına ait **implementation** bolumu sadece modül olarak eklendiğinde derleyici tarafından bulunabilir. Bu sayede ortak bir Trait şablonu ile farklı kütüphaneler arasında derleme öncesinde geçiş yapılabilir. Bu şekilde aktif olarak kullanılmayan özelliğe ait uygulama kodu final derleme çıktısı içerisinde bulunmayacaktır.

3.1.1. Toml formatı

Toml[6] formatı Tom Preston-Werner tarafından tasarlanmış Hash Table veri yapısına birebir olarak yüklenebilen bir veri **serializing** formatıdır. Toml dosyaları hemen hemen her programlama dili tarafından ayrıştırılabilir. Sadece Cargo meta bilgisi için değil Toml adli paket ile Rust uygulamaları tarafından kullanılabilir.

Bu format büyük küçük harfe duyarlıdır. Sadece UTF-8 Unicode karakter kümesinden harfler kullanılabilir. Boşluk için Tab veya Space karakterleri, yeni satir için tek başına 'CR' veya 'CR' ile 'LF' karakterleri birlikte kullanılabilir. Birçok açıdan JSON formatına benzese de Toml insanlar tarafından düzeltilmeye veya görüntülenmeye daha müsait şekildedir. JSON ise insan müdahalesinin olmadığı ve uygulama tarafından yazılan veya idare edilen dosyalar için daha uygundur [22].

Tablo.1. Toml formatının değer türleri

Tür	Açıklama	Örnek (anahtar = değer)
Array	Dizi	baslangic = [0, 0, 85, 170]
Boolean	Doğru yanlış ikilemi	dogruluk = true
Datetime	Tarih	tarih = 1984-01-22
Float	Ondalıklı sayı	sifirdeger = 0.0
Integer	Tamsayı	kontrolno = 43605
String	Yazı dizisi	tablobasligi = "RDST"

[6] https://github.com/toml-lang/toml

Yukarıdaki listeye ek olarak birde tablo türü bulunmaktadır. Tablo türü kok Hash Map tablosu içerisinde tekrarlanan iç içe tablolar koymak için kullanılır.

3.1.2. Kütüphane türleri

Rustc derleyicisi birçok farklı türde çıktı üretir. Önyükleyici gibi alt seviye derlemelerde genellikle tek bir dosya çıktısı yeterli olmamaktadır. Rustc derleyicisine Help argümanı verilerek desteklediği komutlar listelenebilir. Rustc derleyicisine Emit argümanı verilerek derleme çıktısı değiştirilebilmektedir.

Tablo.2. Rustc derleyicisi Emit argümanları

Crate-type türü	Açıklama
asm	Makina kodu
llvm-bc, llvm-ir	LLVM derleyici yapısına ait ara formlar
obj	Obje dosyası
link	Kütüphane paketi
dep-info	Bağımlılık raporu

Rustc Emit argümanı olarak Link değeri aldığında kütüphane çıktısı üretir. Bu kütüphaneler Rlib formatı ile Rust uygulamaları ile Link edilebileceği gibi Staticlib veya Dylib olarak C veya C++ çıktıları ile Link edilebilirler. Önyükleyici tasarımında Asm, Obj ve Link çıktıları bir arada kullanılır. Asm makina kodu olası optimizasyon hatalarını takip etmek veya kod üzerinden çözülmesi güç önyükleyici hatalarını bulmak için kullanılabilir. Unutulmamalıdır ki tezin devamında örnek gösterilen önyükleyici modeli için çekirdek hata ayıklayıcı bulunmayacaktır. Obj yani obje dosyası önyükleyici derlemek fakat Link aşamasına geçilmeden saklamak için kullanılmaktadır. Önyükleyici çalıştırılacağı hedef sistemin kendi Linker uygulaması ile birleştirileceğinden salt objeye ihtiyaç vardır. Dep-info ise kodun bağımlı olduğu diğer kütüphane, obje dosyası gibi gereksinimlerin raporunu tutar. Dep-info bilgisi kullanılarak ile Link aşamasından sonra küçültme işlemi uygulanabilir. Linker her ne

kadar ölü kod optimizasyonu uygulasa da kütüphanelere ait kodlardan hangi kısımların atılabileceği kararı için bu bilgi gereklidir.

Emit argümanı olarak Link kullanıldığında yaratılacak kütüphanenin türü belirlenebilir. Örneğin önyükleyici tasarımında bağımlı olunan kütüphaneler sisteme ait Linker aracına katılacağından bu kütüphanelerin formatı C veya C++ diline ait kütüphane formatı ile uyuşmalıdır. Kütüphane türü Crate-type argümanı ile belirlenir ve tablo ile gösterilmiştir.

Tablo.3. Rust kütüphanelerinin C dilindeki karşılıkları

Crate-type türü	Açıklama	C derleyicisi uzantıları
rlib	Rust statik kütüphanesi	-
dylib	Evrensel dinamik kütüphane	Windows: dll Linux: so OS X: dylib
staticlib	Evrensel statik kütüphane	Windows: lib Linux: a OS X: a

3.2. Dilin özellikleri

Rust semantikleri diğer sistem programlarına göre çok daha açık bir şekilde belirtilmiştir. Öğrenme eğrisinin yüksek olmasına rağmen çok daha az dil tuzağı barındırmaktadır dolaysı ile dili verimli bir şekilde kullanmak büyük bir ustalık gerektirmez. Rust her zaman en güvenli sistematiği kullanır. Örneğin C++ dilinde sıklıkla kullanılan döngü bloklarını döngüsüz forma getirme veya paralelliği arttırmak adına döngü yapılarını alt döngülere bölmek ve sonuçları tekrardan yeni bir döngü ile işlemek gibi alt seviye optimizasyonlara çoğunlukla gerek kalmaz. Bu tip optimizasyonların derleyici tarafından en uygun şekilde yapılamamasının nedenleri arasında otomatik tür çevrimleri veya kopyalama ile eşitleme arasındaki farkın belirgin olmayışı gibi dil tasarımının güvensizliğe sebep olan açıkları sayılabilir.

Rust dilindeki her değerin benzersiz bir sahibi vardır ve bir değişkene bağlı durumdadırlar. Değişkenler ise çalışma aralığı içerisinde bulunmaktadırlar. Çalışma aralığının sona ermesi değişkenin yani kapsadığı değerin silinmesi anlamına gelmektedir [16].

```
1  {
2      let dilim: &'static str = "test1234";
3  }
```

Burada gösterilen dilim isimli değişkenin hedef aralığı tanımlanmasının ardından hemen sona ermektedir. Dilim değişkeni tür olarak bir yazı dizisi dilimidir. String gibi üst seviye yazı türlerinin aksime dilim türü alt seviyedeki verinin hafızadan direkt olarak gösterimine olanak tanır. Bu tür bir adrestir ve burada adresin yasam suresi derleyici tarafından otomatik olarak bulunamayacağından Static yasam suresi belirtilmiştir. Loader tarafından bu uygulama yüklendiğinde dilim değişkeni Static yasam suresi sebebiyle dosya yapısının '.Text' kısmından yüklenir ve çalışma esnasında oluşturulmaz.

3.2.1. Sahiplik

Sahiplik ve ödünç alma sistemi Rust dilinin temelini oluşturan ve belki de teknik olarak en büyük yatırımı almış birimleridir. Sahiplik sistemi sayesinde Rust dili GC kullanmaksızın üst seviye programlama dillerindeki hafıza güvenliğini sağlama imkânı bulur. Ayrıca kopyalama ve taşıma arasındaki ikilemi ortadan kaldırarak performansı arttırma konusunda oldukça başarılıdır. Çünkü değişkenler atandığında kopyalanmak yerine taşınırlar. Buna taşıma semantiği adi verilir.

Tüket isimli fonksiyonu n parametresi ile aldığı değişkeni tüketme özelliğine sahiptir. Çünkü Rust dilinde ödünç verilmeyen her değişken taşıma semantiği gereği tüketilir. Bu durumda 'tuket' fonksiyonu tüketilecek değişkeninin yeni sahibi olmaktadır. Dolaysı ile artık eski değişken kullanmaya devam edilemez.

```
fn tuket(n: i64) { }

let mut tuketilecek = 500;
tuket(tuketilecek);
tuketilecek = 200; // Hata
```

Rust dilinde değişkenler varsayılan olarak değişmezdir. Bu basit gibi gözüken fakat oldukça önemli tasarım birçok kazara yapılan modifikasyonu önlemektedir. Mut anahtar kelimesi ile değişkenler değiştirilebilir yapılabilirler. Ayrıca bir değişkenin adresi alınmak istediğinde bu yine değişmez bir adres sunar. '&Mut' anahtar kelimesi ile değiştirilebilir bir değişkenin değiştirilebilir referansı alınabilir. Fakat değişmez bir değişkenin ise değiştirilebilir referansı alınamaz. Böylelikle adres üzerinden değişmezlikler bozulamaz. Bir değişkenin referansı aynı zamanda onun adresi olarak anılabilir. Çünkü referanslar alt seviyede değişkene ait hafıza bloğunun adresi olarak tutulurlar. Referans üst seviye uygulamalarda yeterli olsa da önyükleyicilerde sürekli olarak güvensiz yazımın kullanılması sebebiyle ham adreslere de ihtiyaç olmaktadır. Ham adresler çoğunlukla referansı alınan değişkenin bir işaretçiye atanması ile kullanılırlar.

```
fn arttir(n: i64) -> i64{ n+1 }

let mut kopyalanacak = 500;
arttir(&kopyalanacak); // Uyari
let _ = arttir(&kopyalanacak);

kopyalanacak = 200;
```

“Genel olarak sahiplik aslında kaynak yönetimidir. Bir kaynak dosya bağlantısı, soket veya birçok farklı şey olabilir, fakat biz sahipliği hafıza yönetimi olarak değerlendireceğiz” [18].

Ödünç alma ise sahibinin değişkeni başka bir değişken veya fonksiyona vermesidir. Örneğin bir bir dosya bağlantısına birçok farklı fonksiyondan yazılma yapılması bu bağlantının bu fonksiyonlara ödünç verilmesini gerektirir. Sonuçta değişken bağlı olduğu kaynağı temsil eder. Kaynağa ait çekirdek Handle adresi, imleç gibi tüm detaylar temsil edilen değişkene erişim bulunduğunda kullanılabilir veya değiştirilebilirler. Arttır fonksiyonuna Kopyalanacak değerini değişmez olarak ödünç verilere bir fazlası hesaplanmaktadır. Daha sonra Kopyalanacak değişkeni değiştirilebilmektedir çünkü Arttır fonksiyonu çalışmasını bitirip döndüğünde aldığı ödünç işlemi de sona ermiştir. Ayrıca Arttır fonksiyonu hesapladığı değerin bir fazla olan sonucu dönmektedir. Rust dili Arttır fonksiyonunun ilk kez çağrılması gibi değer donen fakat sonucu kullanılmayan fonksiyon çağrıları için derleme esnasında uyarı verir ve donen tüm sonuçların kullanılmasını bekler. Küçük büyük harfe şeklen önem veren veri yapıları veya fonksiyon adlarından kullanılmayan değişkenlere kadar Rust dilinin birçok uyarı mekanizması bulunmaktadır. Derleme esnasında hiçbir hata ve uyarının olmaması istenilen sonuca en yakın olunduğu düşünülebilir.

```
1 let mut sayi = 100;
2 let odunc = &sayi;
3 let odunc_2 = &mut sayi; // Hata
```

Değiştirilebilir şekilde ödünç almanın temel kuralı kaynağa münhasır şekilde erişim olmasıdır. Bir değişken sayısız kez değiştirilemez şekilde ödünç verilebilir fakat sadece bir kez değiştirilebilir şekilde ödünç verilebilir. Ayrıca bu değiştirilebilir ödünç alimi çalışmasını bitirip kapsam alanından çıkana yani silinene kadar tekrar değiştirilebilir veya değiştirilemez şekilde ödünç verilemez.

Rust dilinde değişkenler benzersiz ve tek bir sahibi bulunmaktadır. Bu durumda adreslerin farklı değişkenler tarafından paylaşılması olanaksızdır. Bunun yerine veri

yapıları Copy isimli Trait tanımlamasında bulunabilirler. Copy Trait tanımlamasına sahip veri yapıları örneğin değer olarak fonksiyona geçildiklerinde veya eşitleme operatörü ile atandıklarında bit gösterimi aynı olacak şekilde birebir kopyalanırlar [16]. Değerlerin taşınma veya kopyalanması dışında çoğaltılması gerektiği durumlarda vardır bu durumda Clone Trait yapısı ile değerin bir klonu oluşturulabilir ve bu klon başka bir değişkene atandığında yeni değişken artık eski değerin bir kopyasına sahip olmaktadır.

3.2.2. Veri yapıları

Rust veri yapıları temel olarak Struct ve Enum olarak ikiye ayrılırlar. Veri yapılarına ait alanlar varsayılan olarak özel erişim hakkına sahiptirler. Bu basit gibi gözüken kısıtlama tıpkı Mut anahtar kelimesinin kullanım alanı gibi birçok hatalı değişikliği önler. Veri yapıları '#[repr(C)]' temsil satiri ile C dilindeki veri yapılarına denk olacak şekilde gösterilebilirler. Bu şekilde C dili ile etkileşim halinde olan FFI kullanan modüllerde sanki C diline ait veri yapılarıymış gibi hiç bir farklılık olmaksızın kullanılabilirler.

```
1  #[repr(C)]
2  pub struct SmbOrnek {
3      pub tur: u8,
4      pub uzunluk: u8,
5      baglanti: u16,
6  }
7  impl SmbOrnek {
8      pub fn new () -> SmbOrnek {
9          SmbOrnek { tur: 0x17, uzunluk: 128, baglanti: 0 }
10     }
11 }
```

Ayrıca ‘#[repr(packed)]’ temsil satiri ile veri yapısı elemanları hafızada hizalama yapılmaksızın saklanabilirler. Bazı işlemciler erişilecek hafıza adresinin hizalaması açısından sıkı kısıtlamalara sahiptirler. Bu kısıtlama kimi zaman performans kaybına kimi zaman ise işlemci tarafından Alignment Check (#AC) hatası tipinde Interrupt yaratılmasına sebep olurlar [20]. Örneğin X86 mimarisinde Long-mode kipi ile çalışıldığı zaman hafızaya erişim 8-byte ve katları şekilde olmalıdır. Örneğin 1-byte boyutundaki veri yapısı elemanı için 7-byte dolgu verisi konulmalıdır. Bu genellikle derleyici tarafından yapılır veya derleyici veri yapısı elemanlarının yerlerini değiştirerek hafıza alanı kazanmaya çalışabilir. Bu durum önyükleyici tasarımı için problem oluşturmaktadır. Çünkü birçok veri yapısı donanım etkileşimi sebebiyle hafızada gözüktüğü gibi sıralanmalıdır. Bu hizalama doğru yapılmadığı taktirde işlemci bölgeye göre önceki veya sonraki 8-byte bloğu da okuyacak ve kaydırma yaparak istenilen veriyi Register ünitesine kopyalayacaktır. Bu durumda hem fazladan hafıza okunması hem de kaydırma işlemi yapılacağından ciddi bir performans kaybı ortaya çıkacaktır. Fakat SSE gibi paketlenmiş ve ilkel veri türlerinden büyük hafıza bloklarına erisen Instruction komutlarının bazıları doğru hizalanmadığı takdirde işlemci bahsedildiği gibi hata sinyali verecektir.

Struct ilkel veri türlerini, başka Struct yapılarını veya Enum gibi ADT veri yapılarını bir araya getirerek daha komplike yapılar meydana getirmek için kullanılır. Pub anahtar kelimesi ile veri yapısı gizli erişim hakkından açık erişim hakkına çevrilir. Rust dilinde C dilinin aksine veri yapısının elemanları veya kendisinin gizliliği bağımsızdır. Bu örnekte gösterilmekte olan SmbOrnek veri yapısı gizli değildir ve diğer modüler tarafından kullanılabilir. Bağlantı haricindeki elemanlar Pub anahtar kelimesi ile gizli olmaktan çıkartıldıklarından kullanıma açıktırlar.

```
1  mod ornekler;
2  use self::ornekler::*;
3
4  let obje = SmbOrnek::new(); // Yapici fonksiyon ile yaratildi
5  let obje = SmbOrnek { tur: 0, uzunluk: 0, baglanti: 0 }; // Hata
6  let baglanti = obje.baglanti; // Hata
```

Bağlantı elemanı ise sadece yapının Impl bloğu ile belirlenen ve yapıya ait **implementation** kısmında yer alan kurucu fonksiyonalar ile atanabilir. Modül dışında herhangi bir yerden Bağlantı elemanına erişim mümkün olmayacaktır.

Aynı isimli değişkenler tekrar tekrar yaratılabilirler. Buna gölgeleme adi verilir ve sonra yaratılan değişken o noktadan itibaren kullanılmaya devam edilir. Eskisine artık erişim yoktur. Fakat aynı isimli eski değişkenler terk edilemezler ve kapsam dışına çıkıldığında silinirler.

Modüller ise Mod anahtar kelimesi ile var olan modül veya Crate ana dosyası içerisine eklenirler. Eğer Mod anahtar kelimesi Pub gizlilik kaldırıcı anahtar kelime ile birleştirilirse eklenen modül eklendiği modülü ekleyen modüller tarafından görülebilir. Modüle ait Pub anahtar kelimesi ile dışarıdan gözetime acilmiş yapılar Scope operatörü[7] ile kullanılabilir. Use anahtar kelimesi kullanılarak eklenen modüller evrensel kapsam alanına genişletilebilir. Örneğin ‘use self:: ornekler:: SmbOrnek’ ile bu yapı evrensel alana yerleştirilebilir. Eğer modül adından sonraki Scope operatörü ardından yıldız konulursa bu modül içerisindeki tüm kullanıma açık yapıları getirecektir. Buradaki örnekte eklenen modül kullanılan modül içerisine eklendiğinden ilk olarak su anki modülü belirten Self anahtar kelimesi kullanılır. Eğer modül bir üstteki modülden geliyor olsaydı Super**,** pakete ait ana kod dosyasından geliyor olsaydı da Crate isimleri kullanılacaktı.

Mod dosyası ile bir modülün eklenebilmesi için ‘rs’ kod dosyası adinin – örneğin örnekler modülü için – ‘ornekler.rs’ veya örnekler klasörün içerisinde ‘mod.rs’ seklinde isimlendirilmiş olması yeterlidir.

Rust dilinde Enum veri yapısı genellikle fonksiyonel dillerde bulunan ve ADT adi verilen bir sınıfa dahildir [13]. Enum tipli Struct yapısı gibi Repr temsil satiri ile C diline benzer hafıza saklama modeli ile kullanılabilir. Aynı şekilde veri yapısı elemanlarının hafızada kapladığı boyuna göre de temsil edilebilir. Örneğin ‘#[repr(u16)]’ temsil satiri kullanan bir Enum veri yapısının her elemanı ile u16 türünün elemanları (bu boyut işaretsiz 16-bit bir tamsayı ile aynıdır) ile hafızada

[7] Gösterimi yan yana iki tane iki nokta üst üste şekildedir (::)

tutulur. Aşağıdaki örnekte C dili ile tamamen aynı görevi üstlenen ve 3 aşamanın 0-2 kapalı aralığında değerlerini barındıran Enum yapısı verilmiştir.

```
1  #[repr(C, u64)]
2  pub enum Asamalar {
3      Asama1,
4      Asama2,
5      Asama3,
6  }
```

ADT yapıda olması Enum Rust dilindeki türevini diğer sistem programlama dillerindekilerden ayıran en önemli özelliktir. Enum elemanları ilkel türler, Struct veri yapıları veya başka Enum türleri olabilir. Aynı zamanda ikili veya daha fazla elemanın bir araya gelmesiyle oluşan Tuple türleri mevcuttur. Tuple yapıları Struct yapısı içinde olabileceği gibi bağımsız da olabilir. Örneğin 'Struct(i32, i32)' tanımlaması iki tane 'i32' türündeki elemanın bir araya gelmesiyle oluşan Tuple Struct türüdür. Struct veri yapısı ile tamamen aynı özellikleri taşır ve elemanlarının ismi yoktur. Elemanlarına sıfırdan başlayarak sırasına göre erişilebilir. Örneğin '.0', '.1' gibi nokta operatörü kullanılarak elemanların sıraya göre elde edilmesine olanak sağlar. Diğer bir Tuple türü ise '(1, 2)' seklindeki çok ögeli yapıdır ve Struct veri yapısının özelliklerini taşımaz. Değişken adlarıyla oluşturulmuş bir Tuple objesine – örneğin '(x, y)' gibi – atandıklarında eğer eleman sayısı tutarlı ise karşılık gelen elemanlar değişkenlere atanır ve Tuple yapısının elemanları yıkma yöntemi ile elde edilmiş olur.

Rust dili aynı zamanda jenerik türleri destekler. Jenerikler Struct, Enum gibi veri yapılarına veya fonksiyonlara uygulanabilir. ADT sınıfındaki Enum veri yapıları ile jenerikler bir araya geldiğinde ideal bir değer dönüş yapısı oluştururlar. Fonksiyonlar her zaman aynı türde sonuçlar dönmeyebilir. Asil olan farklı türdeki başarılı değerlerden ziyade başarısızlık ve hatta kurtarılabilir başarısızlık – örneğin 'belki' ifadeler – gibi ara sonuçlarında dönüş değeri olarak kullanılabilmesine olanak sağlamasıdır.

```
pub enum AdtEnum<'a> {
    Sayi(i64),
    IkiliYapi(i16, i16),
    Ikili((i32, i32)),
    Mesaj(&'a str),
}

let ornek = (1, 2);
let (x, y) = ornek;

let ornek = AdtEnum::Ikili((x, y));
```

Aşağıda bulunan örnekte gösterilmekte olan Option yapısı Rust standart kütüphanesinden alınmaktadır. T jenerik bir türü ifade eder. Jenerik türler ve yasam sureleri Enum veya Struct türlerinin tanımlandığı satırda üçgen parantezler arasında belirtilirler.

```
pub enum Option<T> {
    Some(T),
    None,
}
```

"Option türü gözcü değer olarak 0 kullanan işaretçiler ile aynı programlama modelini kodlamak için kullanılabilir (örneğin C dilinde değersiz olarak sıfır veya herhangi bir adres bulunmaktadır), fakat bu temelde çok daha güvenli bir bicimde olur. Null işaretçiden ayrılma problemi bu modelin kullanılmasından ötürü ortaya çıkar ve mucidi Tony Hoare tarafından milyar dolarlık hata olarak anılır. Bu sebepten dolayı güvenli Rust onların (Null işaretçiler) kullanılmasına hiçbir şekilde izin vermez" [13].

Option yapısına benzer bir diğer tür ise Result yani sonuç mekanizmasıdır. Tıpkı Option gibi bir jenerik değerin varlığını yada hatayı (Err) kapsar. Option yapısının aksine hatada jenerik bir değerdir. Option ve Result birbirine çevrilebilirler. Some içerisinde bulunan değer Unwrap fonksiyonu çağırılarak elde edebilir. Değer None veya Err olması durumda Panic! ortaya çıkar ve uygulama çöker. Unwrap fonksiyonu sonucun varlığına (Some) veya yokluk (None) veya Hata (Err) olduğu durumlarda ise zaten gidişatı kurtarmanın mümkün olmamasına göre hareket eder.

3.2.3. Trait

Trait mekanizması dilin nesne tabanlı dillerde kullanılan Inheritance metodolojisi olmaksızın benzer bir ara yüzü sağlamasına olanak tanır. Trait sınıflar sadece belirli bir özellik kümesini başka bir sınıfa eklemek için değil aynı zamanda dilin kurallarını uygulamak içinde kullanılırlar. Örneğin anonim fonksiyonlar belirli Trait türlerinden birisine uygun şekilde yazılmalıdır. Bunlar Fn, FnMut ve FnOnce olarak üçe ayrılırlar. Dolaysı ile anonim bir fonksiyonun türü bu üç Trait sınıfından birisi olmak zorundadır. Fn anonim fonksiyonu içerisinde herhangi bir değeri ödünç alan fonksiyonlar çağırılabilir. Mut anahtar kelimesi ile değiştirilebilir durumdaki değerler veya aldığı değeri tüketen fonksiyonlar çağrılamaz. FnMut oldukça benzeri olan Fn anonim fonksiyonuna ek olarak Mut anahtar kelimesi ile değiştirilebilir durumdaki değerleri ve değiştirilemez değerleri alan fonksiyonları çağırabilir. Fakat aldığı değeri tüketen fonksiyonları çağıramaz. FnOnce sadece bir kez çağrılan anonim fonksiyonlara ait Trait yapısıdır. Fn ve FnMut anonim fonksiyonlarına ek olarak aldığı değeri tüketen fonksiyonlarda çağırabilir. Çünkü FnOnce bulunduğu ortamı tüketir.

Anonim fonksiyonları değer olarak kullanmak (örneğin fonksiyon dönüş değeri olarak veya argüman olarak) tek başına mümkün değildir. Bunu yapmanın en kolay yolu Box içerisinde kullanmaktır. Box hafızanın Heap bölümünden ayırma yapan bir veri yapısıdır [15]. Önyükleyici ortamında Heap alanı henüz belirlenmediğinden ve ayırma, silme işlemlerine ait bir hafıza yönetim bulunmadığından Box sistemini kullanmak da mümkün değildir. Ayrıca Box Rust standart kütüphanesinde talimlidir. Yani Heap hafıza alanı yönetimi dilin içine gömülü şekilde değildir.

```
1  trait Dosya {
2      fn oku(&self) -> char;
3      fn yaz(&self, char) -> Result<(), i32>;
4  }
5
6  struct DiskElemaniProxy {
7      fn isaretci() -> u64;
8  }
9
10 impl Dosya for DiskElemaniProxy {
11     fn oku(&self) -> char { 'a' }
12     fn yaz(&self, _: char) -> Result<(), i32> { Err(-1) }
13 }
```

Rust standart kütüphanesi tıpkı her dilde olduğu gibi işletim sistemine bağlı şekilde tasarlandığından önyükleyici ortamında hazır olarak kullanılması mümkün değildir. Örnekte gösterilmekte olduğu gibi Dosya Trait yapısı fiziksel bir dosya sistemi üzerine okuma ve yazma yapabilen basit bir şablondur. Gösterilmekte olan Proxy sınıfı genellikle başka sistem programlama dillerinde kullanılan bir sınıfın derlenip çalışmasını sağlan fakat esas amacı değil de basitçe örnek işlemler yürüten bir tasarım örüntüsüdür. Rust standart kütüphanesinin 'std ::io' bu amaca yönelik bir IO Trait sınıfı bulunmaktadır.

Trait kavramı ara yüz oluşturmak için yegane yöntemdir. Trait kullanan türler statik şekilde sevk edilirler. Statik sevk derleme esnasında uygun türlerin eşleştirilmesi ve sanki ana türün esas elemanlarıymış gibi makina koduna çevrilmesidir. Derlenmiş bir Rust uygulamasında Trait mekanizmasına ait bir kanıt kalmaz. Dolaysı ile Trait yapıları hiç bir çalışma zamanı performans kaybı olmaksızın kullanılabilir [9].

```
1  fn baslat<T: Dosya>(a: T, b: char) {
2      a.yaz(b).unwrap();
3  }
```

Başlat fonksiyonu T jenerik türüne sahip bir fonksiyon olmakla birlikte T türüne ait a argümanı ve karakter türünde bir b argümanı almaktadır. Aynı zamanda T türünün Dosya Trait yapısına bağlı olduğunu belirtmektedir. Eğer bu bağlantı direktifi olmasaydı T türünün Yaz fonksiyonuna sahip olup olmadığı bilinemeyecek ve derleme hatası ortaya çıkacaktı. Where anahtar kelimesi bir türün bir Trait sınıfına bağlı olduğunu ifade etmek için kullanılır. Buradaki örnek Where anahtar kelimesi kullanılarak da yazılabilir. Birden fazla jenerik türünün kullanıldığı veya birden fazla Trait yapısına bağlı olan tür bulunduğu durumlarda görsel açıdan Where anahtar kelimesi kolaylık sağlamaktadır. Aynı zamanda eğer jenerik tür basla bir sınıfın tür parametresi ise bu durumda Where anahtar kelimesinin kullanımı zorunludur. Burada T türü hem Dosya hem de Clone Trait yapılarına bağlı olmak zorundadır. Clone tek bir Clone fonksiyonuna sahip, bir objenin bire bir kopyasının oluşturulmasını sağlayan bir Trait yapısıdır. Rust diline ait sıklıkla kullanılan bazı Trait yapıları derleyici tarafından Impl bloğu kullanılmaksızın türe '#[derive(Clone)]' seklindeki derleyici direktifi ile eklenebilir. Buradaki şart türe ait tüm elemanların da bu direktif ile belirtilen Trait yapısını ekleyebiliyor olmasıdır. Buda tüm elemanların ilkel yapılardan oluşmasını veya Impl bloğu ile belirtilen Trait yapısını eklemiş olması anlamına gelir.

```
1  fn baslat<T>(a: T, b: char) where T: Dosya + Clone {
2      a.yaz(b);
3  }
```

Rust jenerikleri **monomorphisation** yöntemi ile derlenirler. Bu şekilde jenerik yapı ve fonksiyonlarda kullanılan her tür için ayrı ayrı kod yaratılır ve uygun kod blokları kullanıldıkları yerlere yerleştirilir. Bu kod boyutunu arttıran bir yöntemdir ve performans açısından kayıpsızdır [24].

```
1  trait Cevrilebilir<T> {
2      fn cevir(&self, T) -> char;
3  }
4
5  // Burada Where anahtar kelimesinin kullanimi mecburidir
6  fn baslat<T, U>(a: T, b: U) where T: Dosya, U: Cevrilebilir<T> {
7      a.yaz(b.cevir());
8  }
```

Rust dilinde dinamik sevk için sadece Trait objeleri kullanılır. Bunlar Trait yapılarının hafıza üzerinde var olan yani yaratılmış haline verilen isimdir. Dinamik sevkin yapılabilmesi için fonksiyon çağrısının bir işaretçi üzerinden yapılıyor olması gerekir. Tıpkı C++ dilinde mevcut olan Virtual anahtar kelimesi ile objeye ait fonksiyon çalışma zamanında seçiliyorsa dinamik sevk sistem olarak bu metodolojiye oldukça benzemektedir. İşaretçiler 'Box<X>' seklinde Heap hafıza alanında tutulan objeler veya '&X' seklinde referans operatörü ile adresi alinmiş objeler olabilir. Dinamik sevk ile yapılan fonksiyon çağrılarının tümü için bir kez kod yaratılır ve çağrı Vtable adi verilen fonksiyonların adresinin tutulduğu tablodan işaretçiden ayrılma işlemi yapılarak gerçekleştirilir. Tüm çağrılar ortak yaratılmış makina kodu kullandığından dosya boyutu açısından verimli fakat çalışma zamanı açısından kayıpsız değildir. Bu durumda soyutlama derleme zamanında değildir ve çalışma zamanında da dinamik sevkin yapıldığı anlaşılmaktadır. Dördüncü ve son olarak Marker Trait yapıları ile uzantı metotları tanımlanmasıdır. Bu şekilde Rust haricen tanımlanmış türlere metotlar ekleyebilir [9]. Örneğin Trait argümanı alan bir anonim fonksiyona gönderilen parametre için Rust derleyicisi Sized isimli Marker Trait bağlı olma şartını koyar. Çünkü fonksiyona gönderilen argüman yığında veya Register

üzerinde tutulacağından boyutunun biliyor olması gerekmektedir. Marker yapıları dilin kendisine ait Trait yapılarıdır.

3.2.4. Lifetime

Lifetime Rust hafıza güvenliği modelinin önemli bir parçasıdır. Her referans bir Lifetime tanımı bulundurmalıdır. Ham işaretçiler güvensiz sayıldıklarından Rust dili ham adresler ile Lifetime kullanımını şart koşmaz. Örnekte bir fonksiyonun değiştirilebilir şekilde ödünç alinmiş bir argümanına ait Lifetime on ekleri gösterilmiştir. Bu on ekler için kullanılan harf veya kelimelerin özel bir anlamı olmamakla birlikte genel kullanım küçük a harfi ile başlaması ve devam etmesi seklindedir. Argüman olarak alınan değer sonuç olarak döndüğünden yasam sureleri ayindir ve dolaysı ile Lifetime on ekleri aynı harftir.

```
1  fn arttir<'a>(a: &'a mut i64) -> &'a mut i64 {
2      (*a) += 1;
3      a
4  }
```

Lifetime sistemi tüm referansların güvenli olmasının sağlanması esasına dayanır. Buna göre Rust dilinde hiç bir işaretçi silinmiş veya ilk değeri atanmamış şekilde olamaz. Bu şekilde diğer sistem programlama dillerinde karşılaşılan işaretçilerin silindikten sonra veya ilk değeri atanmadan kullanılmaları mümkün olmayacaktır. Sadece ilkel türler için değil sınıflar için de ilk başlatmanın yapılmış olmasına zorlar. Farklı türlere bağlı olunan durumlarda sahip olunan adreslerin ne kadar sure ile geçerli olduğu bilgisi açıkça Lifetime sistemine belirtilmelidir. Rust aynı zamanda RAII semantiğini destekler. Bu objelerin yasam alanı bittiğinde otomatik olarak silinmesini sağlamaktadır. Ayrıca silinmesi gereken tür için özel yıkıcı fonksiyonlar tanımlanmasını da desteklemektedir [15]. Yıkıcı fonksiyonlar Drop Trait yapısının tür üzerine eklenmesi ile gerçekleştirilir. Zorunlu olarak Drop Trait tek bir fonksiyonun eklenmesini gerektirir. İlkel türler, standart kütüphaneye ait türler için Drop hali

hazırda mevcuttur. Aynı şekilde Box ile Heap alanı üzerinden ayrılan hafıza alanını ifade eden değişken yasam alanı bittiğinde belirttiği alan hafızadan silinir. Drop RAII mekanizması ile açılan dosya, soket gibi alt seviye çekirdek objelerinin kapatılması veya silinmesi için kullanılabilir. Aynı zamanda ARC gibi yapıların sayacının azaltılması uygun bir Drop kullanım alanıdır. Bununla birlikte Rust standart kütüphanesi ARC özelliği için hali hazırda Rc sınıfı ile sağlanmıştır. Aynı şekilde Thread çalışma ortamları arasında Arc sınıfı **atomic** seklinde sayacı arttırıp azaltması dolaysı ile kullanılabilir.

```
1  impl Drop for YikilabilirOrnekTur {
2      fn drop(&mut self) { }
3  }
```

Bir işaretçinin Lifetime değeri tutmakta olduğu esas değerden daha kısa veya eşit olmalıdır. Aksi taktirde bu işaretçi geçerli olmayan bir değerin adresini tutmuş olur. Bazı durumlarda karmaşık objelerin yasam alanının derleyici tarafından tahmin edilmesi mümkün olmasa da genellikle ödünç alma sistemi Lifetime on adlarını otomatik olarak yerleştirir. Bu yüzden çoğu durumda Lifetime açıkça belirtilmek zorunda değildir. Buna Lifetime Elision adi verilir. Ödünç alma sisteminin emin olamadığı durumlara iki referans bulundan bir Struct yapı gösterilebilir [1]. Fonksiyonlar için geçerli aynı kurallar Struct yapıları için de geçerlidir.

```
1  struct Ornek<'a, 'b>{
2      x: &'a i64,
3      y: &'a i64,
4      z: &'a i64,
5      w: &'b i64,
6  }
```

Bir Struct yapısının Lifetime suresi eğer referans elemanlara sahipse bu elemanların Lifetime değerinden daha fazla olamaz. Özetle Rust dili güvensiz olmayan her işaretçinin geçerli bir değere karşılık geldiğini garanti eder.

Ayrıca güvensiz olmayan işaretçiler Null değerini alamazlar. Ancak Option ADT yapısı ile Null işaretçi optimizasyonu yapılabilir. Adresin sifir olma durumu None, adrese sahip olma durumu ise Some denkligini gosterir. Bu durumda adres None konumda iken yanlislikla kullanilmasi mumkun degildir. Bu sekilde guvenlik modeli bozulmadan derleyici tarafindan normal isaretci boyutunda saklanacak sekilde optimize edilebilirler.

3.3. Paralel çalışma

Rust dili hafıza güvenliği için kullandığı sahiplik ve ödünç alma modelini **concurrent** şekilde çalışan çalışma ortamında aynı şekilde kullanır. Bu şekilde Rust hafıza modeli için verdiği güvenilirlik garantisinin aynısini çoklu Thread kullanan uygulamalar için de verebilir. Sonuçta bir Thread çalışma ortamı değişkeni değiştirilebilir olarak ödünç almış ise başka bir tanesi ikinci kez ödünç alamaz.

Concurrency için birden fazla yapı bulunmaktadır. Channel farklı Thread çalışma ortamları arasında veri yapısı gönderip almak için kullanılır. Gönderilecek veri yapısı için Send Trait tanımlı olduğu durumda kanal üzerinden gönderilebilir. İlkel türler ve paralel çalışmaya uygun standart kütüphane elemanları için Send Trait zaten tanımlıdır. Aksi taktide kanal üzerinden gönderim yapılamaz. Yani çalışma zamanında **data race** durumu oluşması mümkün değildir çünkü bu güvenlik modeli derleme esnasındadır. Benzer şekilde Sync Trait tanımlı olan veri yapıları Thread çalışma ortamları arasında paylaşılabilir. Aynı anda yapılan okuma ve yazmalar bahsedilen veri yapısında bozulmalara yol açmaz. Sync Trait genellikle ilkel türler ve değiştirilemez şekilde ödünç alinmiş basit veri yapıları tanımlıdır. Ayrıca dile ait RwLock gibi **concurrency** elemanları da bu şekildedir. Kilit sistemiyle çalışan bu mekanizmalar altta yeten veriyi kapsarlar ve veriyi kilitleyerek aynı anda sadece bir Thread çalışma ortamının veriye erişimini sağlayarak bozulmaları engellediğinden Sync Trait bu türler için tanımlıdır. Bu senkronizasyon tıpkı diğer sistem programlama

dillerinde olduğu gibi çalışma zamanında gerçekleşir. Fakat Rust dilinde Mutex, RwLock gibi kilit mekanizmaları kodu değil altta yatan veriyi kilitlemek üzere tasarlanmışlardır [10].

Önyükleyiciler paralel çalışan sistemler değildir ve bu tip bir mekanizmaya ihtiyaç bulunmamaktadır. X86 mimarisi tek bir işlemci çekirdeği ile baslar ve diğer çekirdekler sanki birer fiziksel işlemciymiş gibi işletim sistemi tarafından başlatılır ve Interrupt kontrolcülerin tüm işlemciler için programlanır [20]. Rust dili tüm **concurrency** alt yapısını standart kütüphane aracılığı ile sunar. Böylelikle bu alt yapının kütüphane ile birlikte dilin temelinde değişiklik yapılmadan gittikçe daha fazla önem kazanan paralel sistemlerin daha da gelişmesi ile radikal yenilikler kazanma potansiyeline sahiptir.

4. RUST VE DİĞER DİLLER

Rust dili sistem programlama evreninde temel kurallar olarak kabul edilen ve yıllar boyu zararlara yol açmış tasarımsal hataları sunduğu hafıza güvenliği modeli ile çözmeye yönelik birçok imkan sunmuştur. Aynı şekilde C dili ile oldukça uyumlu bir FFI alt yapısına sahiptir. Rust güvensiz olarak yazılmak için Unsafe anahtar kelimesini kullanır ve FFI bu güvensiz yazımın kullanıldığı en önemli örneklerden birisidir.

4.1. Bilindik problemlerin incelenmesi

Hafıza güvensizliğinde en temel problemlerden ilki dizi üzerinde yapılan indekslemenin dizi boyutundan taşmasıdır. Eğer erişilen dizi elemanı dizi boyutundan büyük olduğu durumlarda yukarı küçük olduğu durumlarda ise aşağı tasma gerçekleşir. Örneğin geçici belleğin dışına yapılan okuma ve yazma işlemleri C dil ailesinde belirsizdir. Yazma işlemi kaçınılmaz olarak veri bozulmasına yol açsa da bu belirsizlik her zaman kendini bir hata olarak göstermez. Çoğu zaman fark edilmez ve uygulamanın gelişmesiyle takip edilmesi çok daha zor hatalara yol acar. Aynı şekilde okunmak istenen adres işletim sistemi tarafından başka bir veri yapısına tahsis edilmiş olabilir. Sinir hatları dışından okuma yapmak güvenlik açısından hat dışına yazmak kadar tehlikeli olmasa da uygulamanın stabil olması açısından oldukça büyük bir risk unsuru olmaktadır.

Dizi indeksinin sinir hatlarının dışında olması problemi ayrıca uzaktan gelen saldırılara da zemin hazırlamıştır. Örneğin X86 mimarisi Call Instruction komutu ile bir fonksiyon çağırdığında bu fonksiyonun tamamlanmasına müteakip geri döneceği Instruction Pointer (32-bit mimariler idin EIP, 64-bit mimariler için RIP olarak adlandırılır [20]) adresini yığında tutar. Ağ üzerinden dizi boyutundan daha büyük boyutlarda yollanan içerisinde saldırı amaçlı makina kodu barındıran veri geçici bellekten tasar ve bu dönüş adresinin üzerine yazar. Böylece fonksiyonun tamamlanması ile ağ üzerinden alınan makina koduna atlanır ve bu kod çalışmaya baslar. Hatta daha gelişmiş bir mekanizma kullanması halinde saldırgan mekanizma kendisine daha büyük bir bellek alanı tahsis eder ve zararlı kodun kalan kısmini buraya indirir. Bu şekilde uygulamanın kontrolünü ve hatta uygulamanın sistem çapında

yetkileri fazla olması durumunda sistemi ele geçirilebilir. Bu tip felaket senaryoları gerçekleşmese dahi sinir hattı dışına tasan yazma işlemleri uygulamanın stabilizesini ciddi anlamda düşürecek ve rastgele bir şekilde çökmesine sebep olabilecektir.

```
let dizi = [1, 2, 3, 4];

// Sorunsuz bir sekilde derlenir
// Calisma zamani sinir hatta tasmasi nedeni ile panic! ile cokme
let sayi_5 = dizi[4];
```

Rust dili dizilerin sınırlarını çalışma zamanında kontrol eder. Dizinin doğası gereği genellikle boyutu veya erişilmek istenen indeksi derleme esnasında belirli değildir. Rust çalışma zamanında sıfır performans maliyetine yol açma iddiasına karsın bu gibi bazı dil bileşenleri çalışma zamanı kontrollerine tabii tutulur. Modern bilgisayar mimarilerinde, gömülü sistemler de dahil olmak üzere kazanılan güvenlik getirilerinin yanında bu maliyet yok sayılabilecek kadar düşüktür. Benzer şekilde diziye negatif indeksler ile erişmek mümkün değildir.

```
let dizi = [1, 2, 3, 4];

// Derleme hatalari
let indeks = -1i32; // Negatif indeks
let indeks = -1 as isize; // Index Trait negatif için tanimli degildir

// Calisma zamani hatalari
let indeks: u64 = 10 – 11; // Isaretsiz tamsayi üzerinde tasma

let sayi = dizi[indeks];
```

Dizi indekslemesi ‘core:: ops:: Index’ Trait yapisi ile gerçekleştirilir ve negatif sayı alabilen türler için tanımlı değildir değildir. Bu durumda dizi negatif indeks alamayacağından sinir hattının aşağısına erişim ancak tamsayı taşması ile olabilir. Rust dilinde işaretsiz türler üzerinde tamsayı taşması da çalışma zamanında kontrol edilir Panic! ile çöküşe neden olur.

Dizi indeksinin sinir dışına taşmasına benzer bir başka hata ise Iterator hükümsüz kılınmasıdır. Iterator yapısı dizi üzerinde indeksleme yöntemine nazaran çok daha güvenlidir. Sadece dizi içerisinde hareket edebilir ve örneğin C++ dilinde Iterator aracılığı ile dizi değiştirilebilir olması gereken durumlarda Iterator yapısında değiştirilebilir olmalıdır. Fakat Iterator aracılığı ile dizi üzerinde dolaşım yapılırken dizinin modifikasyona uğraması Iterator yapısının hükümsüz kılınmasına sebep olur [7]. Bunun sonucu olarak dizinin yapısı bozulabilir veya döngü sonsuz bir bicimde tekrarlamaya başlayabilir.

```
1  let mut vektor1 = vec![1, 2, 3, 4, 5];
2
3  for i in &vektor1 {
4      let yarim = i / 2;
5      vektor1.push(6); // Hata
6  }
```

Örnekte Vec! makrosu kullanılarak bir vektör yaratılmış ve Rust dilinde Iterator yapısının bu problemi önlediği gösterilmiştir. Vektor1 değişkeni Iterator tarafından değiştirilemez şekilde ödünç alındığından Push fonksiyonu ile vektör sonuna yeni eleman eklenemez. Çünkü Push fonksiyonu vektörü değiştirilebilir seklinde ödünç almak isteyecektir [8].

C ve C++ dillerinde makrolar Preprocessor adi verilen bir uygulama ile işlenir. Bu birim makro tanımlarını kullanıldıkları yerdekiler ile değiştirir. Bu basit bir yazı bul ve değiştir işleminden ibarettir. Bunun sonucunda ‘.i’ uzantılı bir dosya çıktısı oluşturur ve bu dosya derleyiciye girdi olarak sunulur. C diline ait makrolar özellikle

aynı makronun birden fazla kere kullanıldığı veya dikkatli kullanılmadığı durumlarda makro cevrimi kolaylıkla değişkenlerin üzerine yazabilir veya mükerrer tanımlama yapabilir. Rust dilinde ise makrolar Hygienic yapılıdır ve AST donuşumunum bir parçasıdırlar [15].

```
1  #define ARTTIR(x) x+1
2
3  int sayi = 8;
4  int sonuc = ARTTIR(sayi << 2); // Hata
5  // sonuc = 64
```

Yukarıda örnek bir C makrosu gösterilmiştir. Arttır makrosu aldığı sayının bir fazlasını dönmektedir. Sonuç değişkenin değeri sayı değişkeninin ikinin kuvveti olması dolaysı ile 2 Bit sağa kaydırarak hızlı bir çarpma işlemi yapılmış ve ardından bu sonucu bir arttırılmış halidir. Fakat C dilindeki operatör öncelikleri sebebiyle 33 olması gereken sonuç 64 olarak gözükmektedir. Bunun sebebi makroya verilen işlemin sonucunun değil kaydırılacak bit sayısının bir arttırılmasıdır. Aşağıda aynı makronun Hygienic Rust makrosu ile yazılmış hali gösterilmektedir.

```
1  macro_rules! arttir! {
2      ($x:expr) => ($x + 1);
3  }
4
5  let sayi = 8;
6  let sonuc = arttir!(sayi << 2);
7  // sonuc = 33
```

Silinmeden sonra kullanım ise hafıza bolumu kullanılmıyor olarak işaretlenir veya silinir ve daha sonra bu bolümde işlem yapılmaya devam edilirse oluşmaktadır. İşletim sistemi şans eseri bu silinen hafıza alanını tekrar kullanıma kazandırmamış olabilir.

Bu durumda bir hata ile karşılaşılmadan devam edilebilir. Uygulama karmaşık hale geldikçe tıpkı dizi sınırları dışına taşma hatası gibi bulunması güç hatalar ortaya çıkabilir. Uygulama silinmiş bir alan üzerinde tekrar okuma veya yazma işlemi uyguladığında eğer sistemi silinen alanı tekrar geri kazandırmış ise Page Fault gibi hafıza hataları ortaya çıkar. Üst seviye dillerde hafıza GC tarafından yönetildiğinden silinmeden sonra kullanım gibi hafıza hataları mümkün değildir. C, C++, Rust gibi sistem programlama dilleri GC kullanmadığından farklı bir çözüm bulunmalıdır. Silinmeden sonra kullanım C veya C++ dilinde çoğunlukla karşılaşılan bir hafıza zafiyetidir. Rust dilinde ise sahiplik sistemi sebebiyle GC seviyesinde bir güvenlik modeli oluşturulmuştur. Hatta GC kullanılan dillerde örneğin kapatılan dosya, soket gibi işletim sistemi objelerinin tekrardan kullanımı mümkündür. Rust bu alt seviye bağlantılar için de aynı hafıza modelini kullandığından daha ileri bir güvenlik modeli ortaya koymaktadır [14].

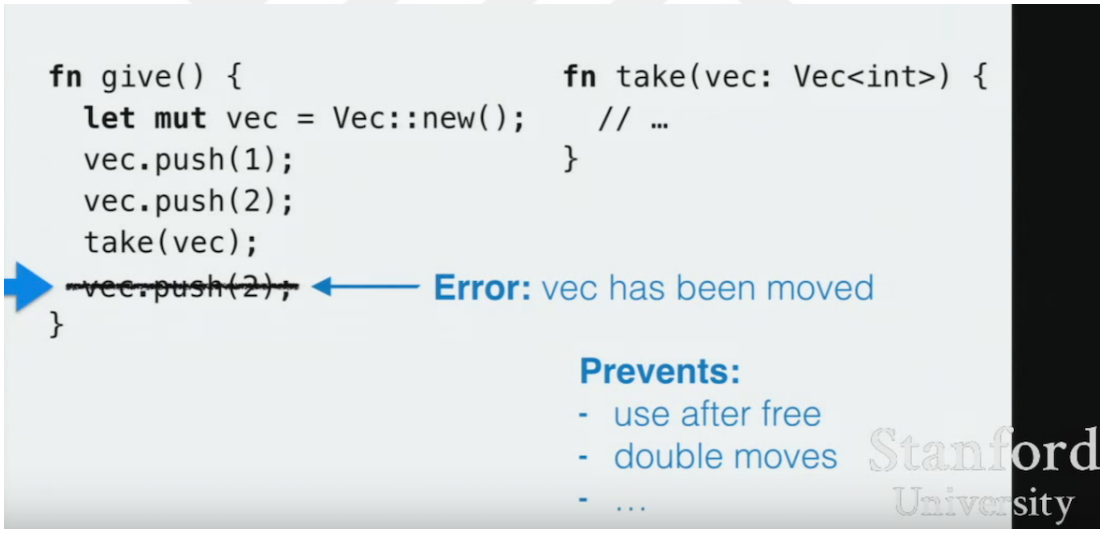
```
1  int* yarat() {
2      int sayi = 100;
3      return &sayi;
4  }
5
6  int* sayi = new int;
7  *sayi = 1;
8  delete sayi;
9  *sayi = 10; // Silinmeden sonra kullanım
10
11 int* sayi_2 = yarat();
12 *sayi_2 = 50; // Silinmeden sonra kullanım
```

Yukarıda gösterilmekte olan C++ diline ait örnekte iki farklı silinmeden sonra kullanım incelenmektedir. İlk olarak yığın üzerinde yaratılan bir değişken fonksiyon dönüş değeri olarak kullanılmıştır. Fonksiyonun çalışmasını tamamlaması ile bu değişkenin yasam alanı sona ereceğinden bu adrese daha sonra ermek silinmeden sonra kullanım hatasına yol acar. Aynı şekilde Heap alanı üzerinde ayrılan sayı isimli

değişkene ait hafıza bolumu Delete operatörü ile kullanım dişi olarak işaretlenmesinden sonra kullanılmaya devam edilmesi mümkündür. Bu durumda bu değişkene erişim yine silinmeden sonra kullanım hatası ortaya çıkmaktadır.

```
1  let sayi = Box::new(5);
2  drop(sayi);
3
4  *sayi = 0; // Hata
```

Rust dili ile sahiplik sistemi sayesinde yığın hataları yapmak mümkün değildir. Benzer şekilde Drop gibi hafıza bölümünü serbest olarak işaretleyen fonksiyonlar aldığı argümanı tüketeceğinden tekrardan kullanımı mümkün olmayacak ve bu derleme esnasında fark edilmektedir.



Şekil.3. Rust dilinde tüketilme örneği

Kaynak: Turon, A. (2015). Stanford Seminar içinde. [8]

4.2. Güvensiz yazım

Rust dilinde güvenli ve güvensiz kod arasında bariz bir çizgi vardır. Güvensiz kod tür sisteminin sağladığı kontrole ve hafıza güvenlik modeline bazı yönlerden tabii değildir. Güvensiz yazım Unsafe bloğu ile gerçekleşir. Direkt olarak bu kod bloğu

kullanılabileceği gibi Unsafe olarak işaretlenmiş fonksiyonlar da kullanılabilir. Güvensiz fonksiyonların çağrılması yine ve ancak güvensiz kod bloğunun içinden yapılabilir. Örneğin türler arası kontrol cevrim yapan Transmute fonksiyonu veya ham işaretçilerin referanstan ayrılması edilmesi Unsafe bloğunu gerektirir [16].

C dili sistem programlamanın temel dilidir. İşletim sistemleri ile etkileşim kurmak isteyen bir uygulamanın C dili kullanması kaçınılmazdır. Üst seviye dillerde bu gerekmese de aslında bu dillere ait standart kütüphaneler veya yardımcı sınıflar aslında C dilinin üzerinde birer soyutlama yapılarıdır. Rust dilinde tasarım ne kadar basarisiz olursa olsun tür, hafıza veya **concurrency** yönünden güvensiz bir işlem yapılamaz [26]. Fakat bazı tasarımlar ideal olmayan koşullar gerektirir. Uygulamalar için genellikle işletim sistemi ile iletişim veya C dili yazılmış kütüphaneleri kullanmak gibi örnekler verilebilir. Önyükleyici, işletim sistemi veya aygıt sürücüsü tasarımında ise güvensizlik kaçınılmazdır. Güvenli ve güvensiz tasarımları kullanan bir Rust uygulaması tamamen güvensiz bir dilde yazılmış türevinden çok daha güvenlidir. Bunun sebebi olası hafıza, tür veya bilinmeyen herhangi bir hatada sorun Unsafe blokları arasında veya buradan dışarıya acılan değişkenlerin kullanımında olacaktır. Böylelikle problem alanı oldukça dar ve tespiti kolay olmaktadır. Aynı şekilde Rust ekosistemine ait geliştirme araçları ve ortamı Rust dili ile tasarlanan uygulama güvensiz dahi olsa aynı şekilde kullanılmaya devam edilebilir.

```
1  let sayi = 8.0f64;
2  // sayi = 8.0
3
4  let yenisayi = unsafe {
5      let y = sayi;
6      let mut x: *mut u64 = mem::transmute(&y);
7      *x = *x ^ (1 << 63);
8      *(x as *mut f64)
9  };
10 // yenisayi = -8.0
```

Yukarıdaki örnekte Bit işlemleri yapılarak F64 türündeki ondalıklı bir sayının negatife çevrilmesi gösterilmektedir. Transmute fonksiyonu kontrolsüz olarak birbirine çevrilebilmesine olanak tanımaktadır. Bit işlemleri ondalıklı sayılar üzerinde tanımlı olmadığından öncelikli olarak 64-bit pozitif bir ham işaretçinin sayı değişkenine referans alması sağlanmaktadır. IEEE-754 ondalıklı sayı standardına göre ilk Bit değerinin işareti göstermesi gerekir. Bu bilgiden yola çıkılarak Bit üzerinde XOR işlemi yapılarak sayının negatifini almak mümkündür. Daha sonra işaretçi 'f64' türüne aitmiş gibi gösterilir ve referanstan çıkartılarak yeni sayı değişkeni içeresinde sayının negatif hali elde edilmektedir.

Ham işaretçiler C dili işaretçileri ile aynıdır ve FFI kullanıldığı durumlarda birbirleri yerine kullanılabilirler. Ham işaretçiler herhangi bir adresi işaret edebilirler. Bu adres geçerli bir adres olmayabilir veya sıfır olabilir. İşletim sistemi bileşenleri için C tipi işaretçiler hayati önem taşır. Çünkü güvenli işaretçiler sanal hafızanın varlığına veya hafızanın dile ait tahsis edici fonksiyonlar tarafından yarabildiği esasına dayanır. Ham işaretçiler üzerinde bir hafıza yönetim mekanizması yoktur. Rust yasam suresi bittiğinde bu değişkenler üzerinde Drop işlemi çalıştırmaz. Aynı şekilde sahiplik ve ödünç alma semantikleri ham işaretçiler üzerinde geçerli değildir. En belirgin kısıtlama '*Mut' ve '*Const' olarak ikiye ayrılmaları ve '*Const' ham işaretçisinin değiştirilemez olmasıdır. Fakat güvensiz olduklarından birbirine çevrilmeleri mümkündür. Lifetime on adları her iki tür için de geçerli değildir ve taşıdıkları adresten uzun veya kısa yasam suresine sahip olmalarının bir önemi yoktur. Kullanım alanları her zaman alt seviye uygulamalar değildir. Çift bağlı liste bunun en güzel örneğidir. İki yöne birden değiştirilebilir referans vermek ödünç alma semantiği açısından uygun değildir ve izin verilmez. Bu durumda ham işaretçilerin kullanılması gereklidir [18].

İşletim sistemi, önyükleyici veya aygıt sürücüsü gibi alt seviye uygulamalar sabit bir hafıza adresine sıkça erişmekledirler. Bu adres fiziksel hafızada gerçek bir adres olabilir. Örneğin Interrupt tablosu hafıza üzerinde sabit bir adreste bulunur. İşlemci işletim sistemine bildiri oluşturacağı zaman kesme türüne göre uygun mesajları Register belleği içerisine koyar ve bu tablo üzerindeki işaretçi adrese zıplar. Aynı şekilde ACPI tabloları hafıza üzerinde sabit bir adreste bulunur. Benzer şekilde alt seviye adres MMIO hafızasına ait bir adres olabilir. Sisteme bağlı bir çok donanım

üzerinde buluna ufak hafızalar aygıt yazılımı tarafından hafıza üzerinde adres aralıklarına denk getirilir. Bu adreslere yapılan erişimler hafıza kontrolcüsü tarafından geçici hafızaya değil bağlı olduğu donanım üzerindeki hafızaya yönlendirilir. Bu şekilde aygıt sürücüleri yüksek performanslı bir şekilde donanıma erişme imkânı bulurlar. Kompleks X86 mimarisinde bu adresler PCI kontrolcüsü[8] ve ACPI tabloları aracılığı ile yakalanır ve işletim sisteminin çalışma suresi boyunca sabit olarak kullanılırlar.

Unsafe kullanımı sadece bir fonksiyonu değil tüm modülü etkiler. Çünkü güvensiz bloktan açığa çıkan verinin güvenliği garanti edilmez. Örneğin Slice formuna cevrim güvensiz yazım gerektirir. Eğer dizinin bir kısmına erişim hata ile sonuçlanırsa veya hatalı bir dizi boyutu sağlanmış ise bu problem dilimin kullanıldığı birçok farklı yerde ortaya çıkabilir. Bu yüzden genellikle soyutlama modülleri kullanılır. Donanım ile etkileşim halindeki olunacağı veya C kütüphanesi ile iletişim kurulacağı zaman bir modül oluşturulur ve bu güvensizliği soyut hale getirir. Güvenli Rust ile kullanılması mumun olmayan dört temel ilke bulunmaktadır. Bunlar ham işaretçilerin referanstan ayrılması, Başka Unsafe veya Instristic fonksiyonların çağrılması, Static yasam suresine sahip değişkenleri değiştirilmesi ve güvensiz Trait yapılarının oluşturulmasıdır. Güvensiz Trait yapılarına örnek olarak Sync ve Send gösterilebilir [26]. Static yasam suresine sahip değişkenler tüm Rust modüllerden ve hatta farklı cevrim üniteleri Link işlemi ile birleştirileceğinden uygulamaya ait farklı dille yazılmış modüller tarafından erişime açıktır. Bu durumda bu evrensel verinin değiştirilmesi güvensiz yazım gerektirir.

4.3. Diğer diller ile etkileşim

C ve C++ dillerini sistem programlama için uygun kılan sıfır performans maliyetli soyutlama özelliğidir. Bu dile ait String veya Vector gibi soyutlamaların kullanan kütüphanelerin makina kodunda yazılması kadar verimli olmasını gerektirir. Modern birçok programlama dili hafıza güvenliği veya Data Race özgürlüğü sunarlar. C ve C++ dilleri hafıza güvenliğine sahip değildirler. Bu diller ile yazılan uygulamalarda

[8] PCI kontrolcüsü üzerinden açığa çıkartılan adreslere BAR adi verilir.

ortaya çıkan hafıza bozulması hataları bilgisayar güvenliğindeki en eski problemlerden birisidir. Bu hatanın temel sebebi C dilinde işaretçilerin veya objelerin hafıza alanları dışından kontrolsüzce değiştirilebilmeleridir. Buna işaretçilerin yasam sureleri dışında gelişen değişimler eşlik eder [27]. Rust tıpkı C diline benzer şekilde çalışma zamanı performans maliyeti olmaksızın çalışır. Bu durumda C dili ile birlikte kullanılabilir ve buna FFI adi verilir. C++ dili operatör Overloading sebebi ile fonksiyon adlarını Link edilecek Translation Unit formundaki obje dosyalarında değişik şekillerde tutar. Bunun sebebi aynı fonksiyon adinin farklı argümanlar ile tekrar tekrar kullanılabilmesidir. Rust ve C dilleri operatör Overloading kullanmamaktadırlar. Birçok **de facto** kütüphane, işletim sistemi API ara yüzleri C dili ile yazılmıştır. Bunun en büyük sebebi C++ dilinden zamansal olarak önce tasarlanmış olmaları olsa bile birçok geliştirici C++ ile tasarlanmış kütüphanelerin C API sahibi olmasını desteklemektedir.

Rust tıpkı C++ dili gibi fonksiyon adlarını değiştirilmiş şekilde derleyebilir. Fakat FFI fonksiyonlarında '#repr[no_mangle]' temsil satiri ile C dili ile denk fonksiyon adlarının yaratılması sağlanabilir. Rust FFI altyapısı çoğunlukla Extern anahtar kelimesinin kullanımına dayanır. Eğer Extern bir fonksiyon deklare edilmiş ise bu C tarafından çağrılabileceği anlamına gelir. Eğer sadece tanımlanmış ise bu da C dili ile yazılmış bir kütüphaneye ait API ara yüzünde bir fonksiyonu işaret eder [29]. Bu tip C ara yüzünü soyutlayan fonksiyonların bir araya gelmesiyle oluşan kütüphanelere Binding adi verilir. Crates.io sitesinde çoğu büyük C kütüphanesi için Binding paketleri bulunmaktadır ve bunlar '-sys' eki ile bitecek şekilde isimlendirilmişlerdir.

```
1  #[no_mangle]
2  pub extern "C" fn bos_dizi(a: i64) -> *const u8 {
3      let _: i64 = a;
4      [u8; 50].as_ptr()
5      // C dilinden cagirildi
6  }
```

FFI fonksiyonları doğası gereği güvensiz olduklarından kullanımları Unsafe bloğu gerektirir. Pub anahtar kelimesi ile yaratılan Translation Unit obje dosyasında fonksiyonun Link edilebileceği belirtilir. Aksi halde farklı modüller tarafından fonksiyon görünür değildir. Saf C dilinde aynı işlem Static anahtar kelimesi ile yapılabilir. Örnekte dönüş türü olarak işaretsiz 8-bit boyuta sahip bir 'u8' karakteri kullanılmıştır. C dilinde bu türe en yakın olarak Char türü bulunmaktadır. Boyut olarak eşleşiyor olsalar da C dilinde Char türünün işareti tanımlı değildir ve işaretli veya işaretsiz olabilir. Extern ile Fn anahtar kelimeleri arasına yazı formunda Calling Convention eklenebilir. Örnekte "C" şekilde gösterilen Convention sisteminde derlenen C kodu nasıl çağrılıyorsa aynı şekilde çağrılacağını ifade eder. Farklı kütüphaneler hatta aynı kütüphanenin farklı fonksiyonları aynı olmayan Calling Convention kullanabilirler. Bu yığına konulan değişkenleri fonksiyonun mu yoksa fonksiyon çağıranın mi Stack Pointer işaretçisini eksilterek sileceğini gösterir. Aynı şekilde X86 mimarisinde fonksiyon donuşu için Ret ve Retn olarak iki farklı Instruction bulunmaktadır. Convention türüne uygun olan kullanılmadığı taktirde yiyin bozulması ortaya çıkmaktadır.

Libc[9] adıyla platforma ait C dili türleri, sabitleri ve fonksiyonları barındıran yardımcı bir kütüphane mevcuttur. Rust dilinin bir parçası olmakla birlikte dile gömülü şekilde değildir. Rust dilinin **decoupled** tasarımı sayesinde dile gömülü değildir ve Crates.io paket arşivi üzerinden Cargo.toml meta dosyasına bağımlılık olarak eklenerek kullanılabilir.

```
1 [dependencies]
2 libc = "*"
```

Harici kütüphaneler Extern Crate satiri ile modüle eklenir ve Use anahtar kelimesi ile kapsam operatörü kullanılarak Pub ile dışarıya çıkartılmış ara yüzler kullanılabilir. Benzer şekilde Lib türündeki kütüphanelerde aynı tanım satiri görmek mümkündür. Bunun sebebi kütüphane olan Lib modülüne ait genellikle örnek bir proje

[9] https://github.com/rust-lang/libc

bulunmasıdır. Bu sadece ara yüzü çağıran göstermelik bir sınıf, test kodu veya kapsamlı bir örnek proje olabilir. Kütüphane tıpkı dışarıdan bir proje tarafından kullanılıyormuş gibi eklenir ve otomatik olarak bir Regression testi ortaya çıkmış olur.

```
1 extern crate libc;
2 use self libc::c_void, c_int;
```

A isimli argüman kullanılmadığından ile kullanılmayan değişken hatasını baskılamak için '_' ismi ile adlandırılmıştır. Aynı şekilde '_' karakteri ile başlayan herhangi bir değişken ismi için bu uyarı baskılanır. Her uyarı sorunsuz bir çalışma zamanı deneyimine tasarımı dehada yaklaştıracağından güvenliği temel ilke edinen Rust dili derleyici uyarları açısından oldukça zengindir. C dilinde aynı şekilde temel Int tamsayı değişkeninin boyutu da tanımlı değildir. Örneğin 64-bit mimari için Int değişkeni alan bir fonksiyona denk gelen örnek farklı platformlar için değiştirilmelidir. Veya A türünün boyutu sabit kabul edilebilir ve C dilinde 'int64_t' türü kullanılabilir. Bu türler dilin standardının bir parçası olmamakla birlikte standart kütüphanenin kütüphaneler tarafından sunulmaktadır. Benzer şekilde Alloca isimli yığın üzerinde hafıza tahsisi yapan fonksiyon örneklenebilir. İşaretçi boyutları sistem dillerinde hafıza veri yoluna erişim boyutu ile aynı olduğundan genellikle birbirleri ile tutarlıdır. Fakat FFI söz konusu olan modüllerde C dili ile işaretçi alışverişi yapılıyor ise Libc kütüphanesinde bulunan türlerin kullanımı faydalıdır. Boyutların eşit olmayabileceği garantisinin dışında Rust dilindeki ham işaretçiler C dili ile fark gözetmeksin kullanılabilir.

```
1 extern "win64" {
2     pub fn c_fonksiyonu(a: *const u8) -> i64;
3 }
```

Rust içerisinden C diline ait bir fonksiyon çağrılacağında ise yine Extern anahtar kelimesi kullanılır ve bu sefer fonksiyon deklare edilmez. Type anahtar kelimesi ile

tanımlama yapılabileceği gibi Extern anahtar kelimesi kullanılarak blok içerisinde birçok fonksiyon tanımlanabilir. ‘#[Link(name=”ckutuphanesi”)]’ tanım satiri örneği gibi fonksiyonlara ait C kütüphanesi ile Link işlemi eğer kütüphane sistemde kurulu ise sağlanabilir. Önyükleyiciler hedef sistem için özel konfigürasyona sahip Linker uygulaması ile birleştirilirler. Rust derleyicisi veya işletim sistemi ile birlikte gelen Linker çapraz derleme için uygun değildir. Dolayışı ile Link tanım satiri sadece üst seviye uygulamalarda kullanılabilir.

Örnekte Win64 Calling Convention kullanımı gösterilmektedir. 64-bit Windows işletim sistemi tarafından kullanılan bu fonksiyon çağırma yöntemi aynı zamanda EFI önyükleme aygıt yazılımı tarafından da kullanılmaktadır. Win64 düzeni fonksiyona katılan ilk dört argümanı Register belleği olan Rcx, Rdx, R8, R9 üzerinde saklar. Sonraki argümanlar yığın üzerinde saklanır. Yığın 16-byte aralıklarla hizalanır çünkü aynı zamanda bu değer işlemci mimarisinde hafıza veri yolu erişim boyutudur. Tarihi C fonksiyon çağırma düzenlerinde olduğu gibi yığına argümanlar ters sıra ile atılırlar. Çalışması tamamlanan fonksiyona ait yığın daha sonra çağıran tarafından temizlenir [30]. Dört argümana kadar olan çağrılar Register belleğinin SRAM donanım ile üretilmesinden dolayı daha hızlı gerçekleşmektedir. SRAM bellek hiyerarşisinde üst (sıfırıncı) seviyededir ve en hızlı depolama ünitesi ile üretilmektedir. Fakat bu hız göz önüne alınacak ise fonksiyon içerisinde yapılan **recursive** çağrılar, yerel değişkenler gibi yığının kullanıldığı diğer etkenler mevcuttur. Ayrıca X86 mimarisi tüm genel amaçlı Register belleklerini, ondalıklı sayılara ait Xmm Register belleklerini ve tarihi ondalıklı sayı yan işlemcisine ait belleği yığına atan Pushad, Fxsave veya Fxstor gibi Instruction komutlarına sahiptir. Bazı derleyiciler tarafından kullanılıyor veya fonksiyon tasarımı bu komutların kullanımını tetikliyor olması ihtimalide düşünülmelidir.

Link tanım satiri ile belirtilen kütüphane sistem içerisinde kurulu değil ve dosya sisteminden geliyorsa Cargo paket yöneticisi ile bu yolu tanımlamak mümkündür. Cargo.toml Build anahtarı ile Rust kodu içeren bir derleme konfigürasyonu içerebilir. Eğer ‘cargo/ .config’ dosyası mevcut ise bu kod olmaksızın belirtilen konfigürasyon dosyası kullanılmaktadır. Bu tablo Package başlığı altına eklenir.

```
1 [package]
2 build = “build.rs”
3 links = “testframework”
```

OS X işletim sistemi Framework adi verilen C diline ait Header dosyaları ile dinamik olarak çalışma zamanında Link edilecek derlenmiş kütüphane çıktısını bir araya getiren paketlere sahiptir. Aşağıda örnek bir ‘cargo/ .config’ dosyası gösterilmektedir ve Framework türünde bir kütüphanenin bulunması sağlanmaktadır. Rustc derleyicisinin argümanları Clang, GCC derleyici argümanlarına benzer olabilir fakat bire bir uyumlu değillerdir.

```
1 [target.x86_64-apple-darwin.testframework]
2 rustc-flags = “-l framework=testframework
3     -L framework=/usr/local/lib/testframework”
```

Target üçlemesi GNU Autoconf araçları ve LLVM derleyici altyapısı tarafından kullanılan derleme yapılacak hedef sistemi belirtir. Çapraz derleme için oldukça yaygın kullanılır. Eğer çapraz derleme yapılıyorsa sisteme ait kütüphanelerin varlığı yeterli değildir. Öncelikle derleyiciye ait yardımcı kütüphanelerin bu Target üçlemesi ile derlenmesi gerekir. Derleyici oluşturduğu obje çıktısını yardımcı fonksiyonları çağıracak şekilde oluşturur. Bu durumda çıktının derleyici yardımcı kütüphaneleri (örneğin Compiler-rt, Libcxx, Libgcc, Libm gibi) ile Link edilmesi gerekmektedir. Rust dili ile birlikte LLVM kullanıldığında her zaman C++ dili ile birlikte kullanılan derleyici yardımcıları kullanılmayabilir. Örneğin Libcxx C++ dili için uygun olduğundan Rust uygulamalarında gerekli değildir. Fakat normal koşullarda en az bir derleyici yardımcı kütüphanesi (örneğin yığın tahsisi hesaplamaları için) gereklidir ve Target üçlemesi ile derlenmiş olması gerekmektedir. Aynı şekilde Rust standart kütüphanesi aynı Target üçlemesi ile derlenmiş olmalıdır. Üçüncü parti kütüphaneler benzer şekilde belli başlı Target üçlemeleri için hazır derlenmiş halde bulunabilirler.

Fakat çoğunlukla bastan derlenmeleri gerekmektedir. Üçlemenin kısımları tire karakter ile ayrılırlar. İlk kısım işlemci mimarisini belirtmektedir. Ayrıca bu kısımda bazı sistemlere ait alt mimari seçenekleri bulunmaktadır. X86 mimarisi için bir alt mimari seçeneği bulunmamaktadır. İkinci kısım üreticiyi belirtmektedir. Genellikle Windows işletim sistemini kullanan donanımlar için PC diğer, Apple şirketine ait donanımlar için Apple, diğer donanımlar için ise Unknown kullanılmaktadır. Bilmediği üretici isimleri için LLVM derleyici alt yapısı isim verilse dahi Unknown üretici ismini kullanmaktadır. Üçüncü kısımda işletim sistemi bulunmaktadır. Bu kısım ayrıca işletim sistemine ait ABI türünü belirtebilir. Örneğin Linux işletim sistemleri ile birlikte genellikle GNU ABI kullanılmaktadır. Ayrıca üçleme dışında Mcpu parametresi ile işlemci adi, Mfpu ile ondalıklı sayı yan işlemcisi adi, Mfloat-abi ile ondalıklı sayılar için kullanılacak Register ünitesi seçilmelidir. Bu parametreler sağlanmadığı taktirde önceden belirlenmiş varsayılan seçenekler kullanılmaktadır [31]. Rust geliştirme ortamı kurulduğu sistemin Target üçlemesine uygun derleyici yardımcı kütüphaneleri, standart kütüphane ile birlikte gelmektedir. Örneğin derleme yapılmak istenen sistem ‘x86_64-unknown-linux-gnu’ üçlemesi ise kaynak sistem burada belirtilen ‘X86_64’ işlemci mimarisi, bilinmeyen donanım üreticisi, Linux işletim sistemi ve GNU ABI olmasa bile Rust geliştirme ortamı bu konfigürasyonu desteklediğinden bu sistem ait kütüphaneler hazır olarak kullanılabilir. Eğer Rust geliştirme ortamı ile Rust geliştiricilerinin sunmadığı bir sistem üzerinde çalışılmak isteniyorsa bu durumda biraz önce anlatılmakta olan altyapıya ek olarak Rust dilinin kendisi de uygun üçleme ile hedef sisteme derlenmesi gerekmektedir.

Tablo.4. LLVM derleyicisi tarafından kullanılan üçleme parametreleri

Parametre	LLVM kısa adi	Örnek mimari
Üçleme 1. kısım	Arch	X86, Arm, Thumb, Mips, vb.
Üçleme 1. kısım	Sub	V5, V6m, V7a, V7m vb.
Üçleme 2. kısım	Vendor	Pc, Apple, Nvidia, Ibm vb.
Üçleme 3. kısım	Sys	None, Linux, Win32, Darwin, Cuda vb.
Üçleme 3. kısım	Abi	Eabi, Gnu, Android, Macho, Elf vb.
Mcpu	Cpu-name	X86-64, Swift, Cortex-a15 vb.
MFpu	Fpu-name	SSE3, Neon
Mfloat-abi	Fabi	Soft, Hard

Kaynak: Clang 3.9 documentation içinde. [31]

Rust dili için **raison d'être** çalışma esnasında performans maliyeti olmaksızın çalışması ve bu sayede GC yapısı kullanmamasıdır. Bu durum C ve Rust dillerinin yine performans maliyeti olmaksızın etkileşim halinde kullanılabilmesini sağlar. Örneğin Rust uygulamasından FFI aracılığı ile bir C fonksiyonunun çağrılması veya Rust fonksiyonunun çağrılması arasında bir performans farkı yoktur.

C diline bir Rust değişkeni gönderildiğinde değişkenin türü ve yasam suresi önemlidir. Değişken yasam alanı sona erdiği için Rust derleyicisi tarafından Drop tetiklemesi yapılır ve otomatik olarak silinir. Fakat eğer bu veri C tarafından kullanılıyorsa otomatik silinme 'mem ::free' fonksiyonu ile kapatılmalıdır. Mem Rust standart ve Core kütüphanelerinde bulunan bir modüldür ve alt seviye hafıza yönetimi fonksiyonları içerir. Örneğin değişkenlerin hafıza üzerinde hizalanması veya boyutunu öğrenmeye yardımcı olan 'size_of' ve 'align_of' fonksiyonları bu modülün bir parçasıdır. Free fonksiyonu Unsafe bloğu gerektirmez çünkü hafıza sızıntısı Rust dilinde bir güvensizlik olarak kabul edilmemektedir. Bu şekilde yasam alanı bittiğinde silinmesi engellenen değişken C kodu tarafından silinmeli veya C kodunun çalışması bittiğinde takip edilerek Rust tarafından yine Mem modülündeki Drop fonksiyonu çağrılmalıdır. Benzer şekilde kilit kullanan senkronizasyon mekanizmaları ile yapılan hatalardan ötürü verinin kilitli kalması, Deadlock durumu, Rust dili için güvensiz yazım değildir.

```
1  extern {
2      pub fn c_fonksiyonu(a: *const u8) -> i64;
3  }
4
5  let a: Box<u64> = Box::new(0);
6  let b: *const u8 = unsafe { mem::transmute(a) };
7  mem::forget(a); // Hafiza sizintisi
8  c_fonksiyonu(b);
```

Rust derleyicisi türleri açıkça belli edilmesine gerek olmadan kullanıldıkları bağlama göre tahmin eder. Buna Type Inference adi verilir. Yukarıda gösterilmekte

olan örnekte Heap alanı üzerinde tahsis edilen bir ‘u64’ türündeki veri C fonksiyonun argüman olarak geçilebilmesi için sahip olduğu adres ham işaretçiye çevrilmiştir. FFI aracılığı ile aktarılan türler Type Inference mekanizması tarafından karmaşıklık yaratabilecek şekilde tahmin edilebildiği görülmüştür. Bu sebeple örnek üzerinde A ve B değişkenlerinin türleri açıkça belirtilmiştir. Bu gösterime Type Annotation adi verilmektedir. C ve C++ dilleri hafıza açısından bir güvenlik modeli sahibi olmadığından FFI güvensiz yazımı aracılığı ile kullanılan hafıza alanları silinebilir, değiştirilebilir veya bozulabilir. Bu dillerdeki hafıza zafiyetleri CWE[10] numaraları ile birlikte tabloda gösterilmektedir. Rust hafıza güvenliği modeli sayesinde tablodaki sorunların derleme esnasında yakalanabileceğini esasına dayanır.

Tablo.5. C/C++ dillerinde hafızayı etkileyen ortak zafiyetler

CWE ID	İsim
119	Buffer hafıza alanının sınırları ile alakalı işlemlerin doğru olmayan şekilde sınırlandırılması
120	Girdinin boyutuna bakılmaksızın tampon bellek kopyalaması yapılması (Klasik Buffer taşması)
125	Out Of Bounds okuma
126	Buffer Over-read (Heartbleed hatası)
122	Heap tabanlı Buffer tasması
129	Dizi indeksinin doğru olmayan şekilde kontrol edilmesi
401	Son referans kaldırılmadan hafızanın uygun olmayan şekilde silinmesi (bellek sızıntısı)
415	İki defa silme
416	Silinmeden sonra kullanım
591	Uygun olmayan şekilde Lock edilmiş hafızada kritik verinin saklanması
763	Geçersiz işaretçi veya referansın silinmesi

Kaynak: Getreu, J. (2016). Enhance Embedded System Security With Rust (Revision 1.1) içinde. [27]

[10] https://cwe.mitre.org

4.4. EFI teknik şartnamesi

Sofistike işletim sistemi çekirdekleri ve aygıt sürücüleri ile modern X86 mimarisine güncel sistemlerde BIOS tarihi sistemlerin aksine bir önyükleyici temeli görevi üstlenir hale gelmiştir. İşlemciyi Real-mode kipinde ve 1MB hafıza adreslemesi ile başlatması ve farklı donanım üreticileri tarafından üretilen sistemlerde aynı BIOS Interrupt komutlarının farklılık göstermesi gibi problemler bulunmaktadır.

EFI teknik şartnamesi Intel tarafından 1999 yılında BIOS aygıt yazılımına alternatif olarak çıkartılmıştır. 2005 yılından itibaren UEFI[11] adıyla kar amacı taşımayan UEFI Forum tarafından geliştirilmesi devralınmıştır. UEFI 2.0 surumu 64-bit desteği sağlamaya başlamış ve UEFI Forum tarafından yayınlanan ilk surumdur ve atası olan EFI sistemine oldukça benzemektedir. Geliştiriciler arasına birçok katılım gerçekleşmiş ve 2010 yılında katılımcı kurumların sayısı 160 rakamına kadar yükselmiştir. Tarihi BIOS sistemlerde aygıt yazılımı üreticilerin ticari bir ürünüydü. Modern UEFI mimarisi ağ desteğinden insan ara yüz aygıtlarına kadar birçok donanımı aygıt sürücüsü modeli ile desteklemektedir [2]. Varsayılan aygıt sürücülerine ek olarak EFI aygıt yazılımı için aygıt sürücüsü geliştirmek mümkündür.

EFI önyükleyici tasarımı tipik bir uygulama tasarımından farklı değildir. EFI bir işletim sistemi gibi hafızayı başlatır, Interrupt tablolarını oluşturur ve kendisine özel aygıt sürücülerini hazır hale getirir. Uygulamalar ve sürücüler için C dili ile sunulmuş API ara yüzleri mevcuttur. Rust diline ait FFI ara yüzü ile EFI aygıt yazılımına ait API tam anlamıyla kullanılabilir. EFI uygulamaları Windows işletim sisteminin kullandığı PE dosya formatını kullanır dolayısı ile EFI obje dosyaları PE formatında yürütülebilir çıktı üretebilen bir Linker aracılığı ile Link edilmelidir.

UEFI teknik şartnamesi önyükleme, çalışma zamanı ve dosya yükleme protokollerini zorunlu kılar. Bununla birlikte birçok opsiyonlu modül tanımlamaktadır. Zorunlu temel modüller hafıza yönetimi, zamanlayıcılar, olaylar, UEFI değişkenleri, kapsüller ve kod yükleme gibi bileşenleri içermektedir. UEFI aynı zamanda GPT desteğine sahiptir. GPT tablo türü Tarihi sistemler tarafından kullanılan

[11] http://www.uefi.org

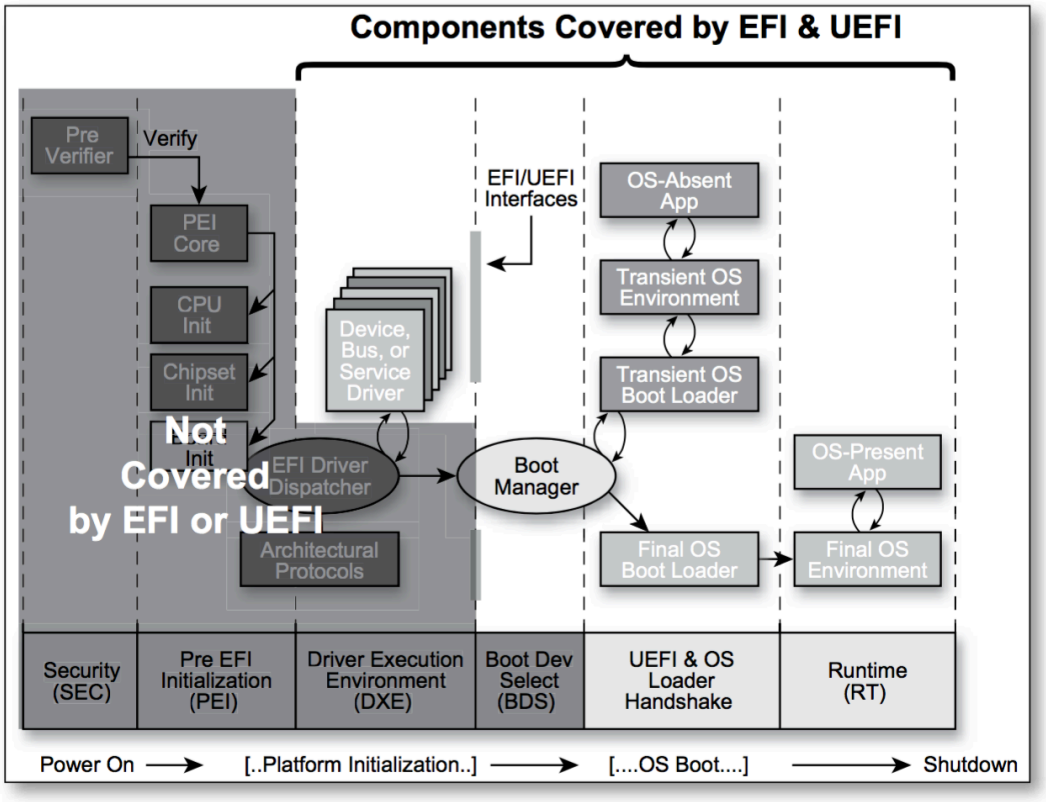
MBR bölümleme tablosuna göre 2 Terabyte bolum sınırlaması gibi kısıtlamalara sahip değildir. UEFI resmi olarak bir uygulama koduna sahip değildir ve sadece teknik şartnameden ibarettir. EDKII[12] isimli açık kaynak bir proje mevcuttur ve UEFI önyükleyici tasarımında referans olarak kullanılabilmektedir. UEFI aynı zamanda Shell adi verilen bir komut istemine sahiptir. Aygıt yazılımı güncellemeleri ise Capsule adi verilen paketler ile yapılır. Sürücülerin tekil olarak güncellenmesi mümkündür ve güncelleme esnasında temel fonksiyonlar kullanılabilir. Bu sayede örneğin I2C veri yoluna yazılarak yol üzerindeki bir donanım aygıt yazılımda güncellenebilir. Tarihi BIOS EEPROM güncellemelerin aksine Capsule dosyaları doğruluk testine tabii tutulurlar [32]. Genellikle aygıt yazılımı tarafından sisteme bağlı durumdaki ortamlarda UEFI uygulamasının başlatılması basarisiz olduğunda Shell çalıştırılır veya direkt olarak başlatılabilir. Çeşitli Shell komutları önyükleyici uygulaması tasarımda yardımcı niteliktedir. Örneğin Memmap komutu hafıza haritasını, Pci sisteme bağlı PCI aygıtlarının konfigürasyon uzayını veya Getmtc su anki Monotonic sayaç değerini ekrana yazdırır. UEFI değişkenleri eğer bozulmaya uğramışlar ise Shell üzerinden görülebilir veya değiştirilebilirler.

UEFI aygıt yazılımı (eğer konfigürasyon menusu mevcut ve aksi şekilde ayarlanmadığı taktirde) FAT32 dosya sistemine sahip ortamlarda ‘efi/ boot/ bootx64.efi’ dosyasını arar ve başlatır.

Hafıza yönetiminin en önemli getirisi hafıza haritasının alınabilmesidir. X86 mimarisinde hafıza MMIO, ACPI tabloları ve bu altyapıya ait veri yapıları, Chipset konfigürasyonu için ayrılmış alanlar ve EFI önyükleme ve çalışma zamanı veri yapıları gibi birçok donanım tarafından tahsis edilmiş bolüme sahiptir [12]. Bu bölümlerden bazıları taşınabilir veya tekrardan kullanılabilir. Örneğin önyükleyici görevini tamamladıktan sonra EFI önyükleyici tarafından kullanılan hafıza tekrardan işletim sistemi tarafından adreslenebilir. Fakat bu alan çekirdeğe ait ise veya çekirdek tarafından hafıza yüklenmesi güç olan bir sürücünün yüklendiği alan ise EFI önyükleyicisine tahsis edilmiş bir alan dahi olsa tekrardan kullanılamaz. Aynı şekilde ACPI tablolarına ait veri yapıları taşınabilir. Fakat Interrupt kontrolcüsü üzerindeki hafızanın adreslediği alanlar taşınamaz. Bazı taşınabilir bolümler masaüstü X86

[12] http://www.tianocore.org/edk2/

sistemlerde az yer kapladıklarından tekrardan kullanılmayabilir ve bu alanların taşınması Fragmentation problemini azaltmak adına veya tekrardan kullanılabiliyorsa işletim sistemi tarafından tahsise açılması gömülü sistemlerde düşünülebilir.



Şekil.4. EFI aygıt yazılımının kapsadığı bileşenler

Kaynak: Zimmer, V., Rothman M. ve Marisetty S. (2011). Beyond BIOS içinde. [2]

UEFI değişkenleri kalıcı hafızada tutulurlar ve sistemde başka bir kalıcı depolama birimi olmasa dahi kullanılabilirler. Önyükleme sırasında çekirdek güncellemesi bilgisi veya işletim sistemi lisans sertifikası gibi bilgiler burada saklanabilir. UEFI çalışma zamanı modülü ise çekirdek yüklemesi tamamlanıp önyükleyiciden çıkış işlemi gerçekleştikten sonra işletim sistemi çalışması esnasında sistem saati, güç kaynağı komutları veya Monotonic sayaçlar gibi yardımcı fonksiyonları sağlar. Son zamanlarda donanıma fiziksel erişim aracılığı ile yapılan saldırılar bazı UEFI modüllerinin kullanımını tartışma konusu haline getirmiştir.

4.5. EFI platform yazılımı

UEFI yazılım mimarisi protokollere ayrılmıştır. Her protokol kendine ait GUID dizisi ile ayırt edilir. Örneğin dosya sistemine erişim protokolü olan Load_File_Protocol teknik şartname tarafından zorunlu olarak eklenecek şekilde tanımlanmıştır. UEFI önyükleyici uygulaması başladığında giriş fonksiyonu argüman olarak Efi_Handle türünde var olan imajı ve Efi_System_Table türünde Boot Services, Runtime Services ve Configuration Table işaretçilerinin bulunduğu bir veri yapısı ile başlatılır.

Tablo.6. Minimal önyükleme için EFI fonksiyonları

Eleman adi	Türü	Bulunduğu tablo
LocateProtocol	Fonksiyon	Boot Services
GetMemoryMap	Fonksiyon	Boot Services
ExitBootServices	Fonksiyon	Boot Services
ConfigurationTable	Dizi	System Table
NumberOfTableEntries	Uint	System Table

Boot Services tablosu kullanarak LocateProtocol fonksiyonu ile uygun GUID sağlanarak istenilen protokole ait Handle objesi elde edilir. Handle objesi bir işaretçiden oluşan tıpkı işletim sistemi API ara yüzlerindeki çekirdek objeleri gibi bir veri yapılarıdır. Eğer protokol birden fazla kısımdan oluşuyor ise (örneğin dosya sistemi protokolü bölümleme tablosu ile birden fazla mantıksal veya fiziksel bölümlere ayrılmış olabilir) HandleProtokol fonksiyonu ile dizi elde edilmesi mümkündür. Yine aynı tabloda hafıza haritası elde edilmek için GetMemoryMap ve önyükleme işlemi tamamlandığında çağrılmak üzere ExitBootServices fonksiyonları bulunmaktadır. ExitBootServices çağırıldıktan sonra Boot Services tablosundan herhangi bir fonksiyon kullanılamaz ve sadece Runtime Services tablosu tarafından sağlanan işlemler yapılabilir. Ayrıca UEFI önyükleyici işlemleri için ayrılan hafıza bu fonksiyonun çağırılmasından sonra işletim sistemi tarafından tekrardan tahsise açılabilir. ConfigurationTable dizisi içerisinde istenilen tabloya ait GUID değeri aranarak ACPI, SMBIOS veya MPS tablolarına ait adresler elde edilebilmektedir. Dizideki eleman sayısı C dilinde ‘Uint’ türü ile NumberOfTableEntries değişkeninde saklanmaktadır. Rust dilinde 64-bit mimaride bu ‘u64’ türüne denktir. Fonksiyonlar

ise Win64 Calling Convention kullanır ve Extern anahtar kelimesi ile Rust fonksiyonları olarak çağrılabilirler. ConfigurationTable dizisi eleman sayısının da bilinmesi üzere Rust dilinde Slice türüne çevrilmek için uygun bir adaydır. Bu sayede Iterator yapısı ile dizi üzerinde dolaşmak mümkün olmaktadır. Iterator türüne ait tüm yüksek mertebe fonksiyonlar bu Trait yapısını tanımlayan her tür tarafından kullanılabileceğinden Slice cevrimi tüm bu fonksiyon kümesini otomatikman sağlamaktadır.

Tablo.7. Minimal önyükleme için EFI protokolleri

Fonksiyon adi	Bulunduğu protokol
OpenVolume	EFI_SIMPE_FILE_SYSTEM_PROTOCOL
Open	EFI_FILE_PROTOCOL
Read	EFI_FILE_PROTOCOL
Close	EFI_FILE_PROTOCOL

Uygun protokol fonksiyonları kullanarak Simple_File_System_Protocol tarafından sağlanan OpenVolume fonksiyonu kullanarak ilgili dosya sistemi açılabilmektedir. FAT32 dışında kalan dosya sistemleri yeni UEFI sürücü tasarımları ile mümkün olmaktadır. Açılan dosya sistemi File Handle aracılığı ile çalıştırılabilecek dosya işlemleri Rust dilindeki IO işlemlerine benzerdir. Bu protokol Rust dilince ‘std:: io’ Trait yapısını tanımlayacak şekilde tasarlanabilir. Ayrıca FileInfo fonksiyonu kullanılarak dosyanın boyutu elde edilebilmektedir. Dosya boyutu, eğer çekirdek PE dosya formatında ise okunan ham dosya verisinin formata doğru eslenebilmesi için gerekli olabilmektedir.

4.6. PE dosya yapısı

EFI önyükleyici uygulamaları PE dosya formatında saklanmaktadırlar. İşletim sistemi çekirdeği örneğin ELF gibi Unix işletim sistemi formatı veya herhangi özel tasarlanmış bir formatta olabilir. Eğer önyükleyici ile birlikte test çekirdeği hazırlanıyorsa PE dosya formatını kullanmak uygun olabilir çünkü hali hazırda EFI geliştirme ortamı mevcuttur.

PE dosya formatı Windows işletim sistemi tarafından yaygın olarak kullanılmaktadır. Dinamik kütüphaneler için Dll uygulamalar için ise Exe uzantısına sahip 32-bit ve 64-bit uyumluluğu bulunan bir yürütülebilir bir dosya standardıdır. PE dosyaları Loader tarafından yüklenirken fiziksel veya sanal adresler kullanılabilir [33]. Önyükleyiciler sanal hafıza ve karmaşık Loader yapıları kullanmadığından çekirdeğin başlatılabilmesi için PE formatına ait AddressOfEntryPoint alanında belirtilen başlatma fonksiyonuna ait adres yeterlidir. Önyükleyici bu fonksiyonu uygun Calling Convention ile çağırabilir. ImageBase alanı PE dosyasının yüklenmesi gereken sanal adresi belirtir. Eğer PE çekirdek dosyası bu belirtilen alana yüklenir ise Relocation işlemi yapılmadan çalıştırılabilir. Aksi halde Relocation tablosu kullanılarak adresleme kullanılan tüm Instruction komutları uygun şekilde yer değiştirilmelidir. PE dosyaları Section adi verilen bölümlere ayrılmıştır ve çekirdeğin aslı veya bir sonraki etap çekirdek Loader uygulaması olmasına göre dosyanın ham verisini taşıyan bu Section bölümleri hafızaya kopyalanır. Kullanılan Linker uygulamaları yaratılan imaj açısından farklılık gösterebilir dolayısı ile çekirdek tasarımı ile tutarlı olan kullanılmalıdır.

5. ALT SEVİYE RUST

Rust standart kütüphanesinin oldukça güçlü ve modern yapısı sayesinde çekirdek geliştirme için oldukça uygundur. Aynı zamanda en alt seviyede standart kütüphane olmadan çalışmak mümkündür. Rust standart kütüphanesi Std ve Core olarak ikiye ayrılmaktadır. Std üst seviye dillerden beklenecek kadar kapsamlıdır ve C dilinin üzerine Rust dili standart kütüphanesi beklide en belirgin gelişmelerden birisidir [15]. Core işletim sisteminden bağımsız şekilde tasarlanmıştır dolayısı ile aygıt sürücüsü ve işletim sistemi çekirdeği gibi alt seviye uygulamalarda kullanılabilir. Std ise Rust geliştirme ortamının bulunduğu işletim sistemine bağlıdır. Çapraz derlenebilir fakat işletim sistemi olmadan çalışması mümkün değildir. Karmaşık önyükleyiciler kendi alt seviye donanım etkileşim kütüphanesi üzerine Rust Std yapısını kısmen tanımlayabilir ve daha üst seviye tasarımlar yapılabilir.

5.1. Standart kütüphaneyi kaldırmak

Önyükleyici uygulaması için öncelikli olarak Rust standart kütüphanesinin kaldırılması gerekir. Rust dili ‘No_std’ tanım satiri ile bunu mümkün kılmaktadır. Derleyici altyapısının ihtiyaç duyduğu ‘eh_personality’ ve ‘panic_fmt’ isimli iki fonksiyon bir Rust uygulamasında deklare edilmiş olmalıdır. Standart kütüphane bunu gerçekleştirdiğinden önyükleyici uygulaması boş olarak bunları tanımlar. Uninimplemented makrosu ile fonksiyonun herhangi bir işlem yapmadığını belirtmek için kullanılmaktadır. Bu durumda sadece Core standart kütüphanesi kullanılabilir.

```
1  #![no_std]
2  #![feature(lang_items)]
3
4  use core::*;
5  #[lang = "eh_personality"] extern fn eh_personality() { unimplemented!(); }
6  #[lang = "panic_fmt"] fn panic_fmt() -> ! { unimplemented!(); }
```

5.2. Hedefe derlemek

Core standart kütüphanesinin kullanılabilmesi için öncelikle çapraz derleme yapılması gerekmektedir. Core standart kütüphanesi derlenebilmek için sadece Memcpy, Memset ve Memcmp fonksiyonlarına ihtiyaç duyar[13]. Bu fonksiyonlar C standart kütüphanesinde bulunan hafızadan hafızaya ham taşıma, atama veya karşılaştırma görevi üstlenirler. C dili ile geliştirilmiş türevleri kullanılabileceği gibi Rust dilinin mimarlarından Alex Crichton tarafından tasarlanmış Rlibc kütüphanesi bu üç fonksiyonun Rust dili ile geliştirilmiş sürümünü bulundurmaktadır[14]. Bu sebepten ötürü Rlibc kütüphanesi de Core kütüphaneye ihtiyaç duymaktadır. Bu hafıza fonksiyonları ayrıca #![feature(asm)] tanım satırıyla açılan 'Asm!' makrosu ile X86 makina kodu ile deklare edilebilirler. Örneğin Memcpy fonksiyonu için X86 mimarisinde bulunan Stosb Instruction komutu Byte boyutunda kopyalama yaptığından kullanılabilir. Rep on eki aldığında Byte sayısı RCX, kaynak RSI ve hedef RDI Register birimlerine konulur ve kopyalama sağlanır. Ayrıca Movntdqa Instruction komutu ile 64-bit bloklar halinde kopyalama Non-temporal şekilde yapılabilir. Bu durumda kopyalanan veri Cache tampon belleğine konulmayacak ve Cache kirlenmesi ortaya çıkmayacaktır. Sadece işlem boyutu 64-bit katı olmadığında son blok için bir uç senaryo ortaya çıkmaktadır.

Önyükleyicinin çalışacağı hedef işletim sistemine sahip olmadığından önyükleyici uygulaması, Rlibc kütüphanesi ve Core standart kütüphanesi özel bir Triplet üçlemesi ile derlenmelidir. EFI önyükleyici uygulaması PE dosya formatı kullandığından üçlemenin adi 'x86_64-pc-windows-gnu-custom' şekilde adlandırılabilir. Üçleme adi zaten kullanıldığından Custom eklenerek uç bir senaryoda tasarım yap ildiği gösterilmektedir. Rustc derleyicisi altyapısı tarafından tanınmayan hedef üçlemeleri JSON formatında bir şablona uygun şekilde hazırlanır. Var olan üçlemelerden farklı olarak LLVM derleyici altyapısına ait derleme opsiyonları değiştirilmektedir. 'No-compiler-rt' anahtarı ile derleyici yardımcı kütüphanesi devre dişi bırakılır. Önyükleyici uygulaması doğası gereği bu kütüphaneye ihtiyaç duymayacak ve aksi halde derleme işlemi oldukça karmaşık bir duruma gelmektedir. Benzer bir şekilde 'Elimiate-frame-pointer' anahtarı ile fonksiyon çağrılarında yığın işaretçisinin RBP

[13] https://doc.rust-lang.org/core
[14] https://github.com/alexcrichton/rlibc

Register biriminde saklanması kapatılmış olur. Bu özellik her hâlükârda Win64 Calling Convention ile kullanılmamaktadır. Varsayılan yığın dolduğunda genişletilememesi için 'Morestack'[15] anahtarı kapatılmaktadır. Önyükleme ortamında tasarıma göre ilk etapta veya hiç bir zaman yeni yığın tahsis sistemi olmayabilir.

```
{
"cpu": "x86-64",
"llvm-target": "x86_64-pc-windows-gnu",
"os": "none",
"arch": "x86_64",
"target-endian": "little",
"no-compiler-rt": true,
"disable-redzone": true,
"eliminate-frame-pointer": true,
"morestack": false,
"features":"-mmx,-3dnow,-3dnowa,-avx,-avx2,-sse,-sse2,
        -sse3,ssse3, -sse4.1,-sse4.2",
"pre-link-args": ["-m64"],
"target-pointer-width": "64",
"target-word-size": "64"
}
```

Bir diğer derleyici temelli anahtar olan 'disable-redzone' ile fonksiyon çağrıları arasında yığın işaretçisi işlemlerini azaltmak adına lokal değişkenler için ayrılan boş alan kapatılır. İşletim sistemi bileşenleri mümkün oldukça bu tip optimizasyonlardan uzak durmaktadır.

Son olarak Features anahtarı ile LLVM derleyici altyapısına ondalıklı sayılara ait işlem yapılmaması sağlanır. Ondalıklı sayı sistemi önyükleyici ve çekirdek gibi en alt seviye uygulamalarda bozulmalara yol açabilmektedir. En basitinden büyük boyutta olan 8087 yan işlemcisi veya SSE gibi Register birimleri büyük boyutlu olduğundan

[15] http://gcc.gnu.org/wiki/SplitStacks

kritik zaman aralıklarında çalışması gereken Interrupt Handler fonksiyon çağrıları arasında yığında saklanamazlar [11]. Ayrıca önyükleyici başladığında ondalıklı sayı yan işlemcisi hazır duruma getirilmemiş olabilir. Bu genellikle işletim sistemi çekirdeği tarafından gerçekleştirilir ve önyükleyicilerde ondalıklı sayılar kullanılmazlar. FPU birimi ile birlikte 3dNow gibi matematiksel yan işlemciler de kapatılmaktadır. Fakat Core kütüphanesi içerisinde ondalıklı sayı kullanan yapılar bulunduğundan LLVM derleyici altyapısı bu özellik kapatma işlemine müsaade etmemektedir[16]. Normal şartlarda Rust kaynak kodu içerisinde 'src/libcore' içerisinde bulunan Core kütüphanesi çekirdek geliştirme amacıyla ancak ondalıklı sayı desteği silinerek kullanılabilir [17].

```
1 rustc --emit=obj,dep-info,asm \
2 -O src/main.rs --target=x86_64-pc-windows-gnu-custom \
3 --out-dir=build -Z no-landing-pads -C debuginfo=2 \
4 -L libcore/target/x86_64-pc-windows-gnu-custom/release \
5 -L rlibc/target/x86_64-pc-windows-gnu-custom/release
```

Bağımlılıklarının derlenmesi tamamlanan 'src/main.rs' klasöründe bulunan EFI önyükleyici uygulaması Rustc derleyicisi ile derlenebilir. Arından Core standart kütüphanesi ve EFI uygulaması GNU Binutils[18] 'x86_64-efi-pe' Linker uygulaması ile tek bir obje dosyasına Link edilmektedir. Son olarak da bu imaj pozisyon bağımsız adresleme kullanılarak PE başlığındaki 'Subsystem' değeri 10 olacak şekilde başlangıç fonksiyon adi verilerek EFI imajına link edilmelidir. Ayrıca Binutils paketinin içine önyükleyici tasarımına derleyici obje çıktıları ve Linker imaj çıktılarının incelenebilmesi için birçok araç bulunmaktadır. Final imaj 'bootx64.efi' olarak adlandırılır ve FAT32 dosya sistemi üzerinden önyüklemeye hazır durumdadır. Rustc derleyicisi ile üretilen Asm makina kodu göz ile hata ayıklama makastı ile kullanılabilir. Ayrıca Dsm bağımlılık bilgisi kullanılarak obje üzerinde ölü kodların atılması ile küçülme uygulamak mümkündür. Rustc derleyicisine '-C debuginfo'

[16] https://github.com/rust-lang/rust/issues/26449
[17] Philipp Oppermann tarafından gerçekleştirilmiş bir çalışma mevcuttur https://github.com/phil-opp/nightly-libcore
[18] https://www.gnu.org/software/binutils

argümanı ile hata ayıklama sembollerini yaratması söylenmektedir. Çekirdek hata ayıklayıcısının EFI uygulaması için tasarlanması halinde bu semboller kullanılarak hata ayıklaması yapılabilir. Son olarak '-Z no-landing-pads' argümanı ile Panic! desteği ve olası hata durumunda yığını temizleyen Unwinding mekanizması kapatılmaktadır. Önyükleyicide kritik hata oluşması durumunda bu durumu kurtarıp devam edilmesi mümkün değildir.

5.2.1. Test ortamı

Linker tarafından üretilen EFI uygulaması imajı donanım üreticisinin belirlediği önyükleme ayarları ile USB veya CD depolama aygıtı aracılığı ile test edilebilir. Ayrıca sanal makina üzerinden test etmek daha kolaydır. Qemu[19] sanal makinesi X86 mimarisini destekler ve harici BIOS desteği bulunmaktadır. Gömülü olarak EFI desteği bulunmadığından açık kaynak kodlu olan EdkII[20] aygıt yazılımının Qemu cevrimi olan Ovmf[21] aracılığı ile Rust ile yazılmış bir önyükleyicinin test edilebilmesi mümkündür. Qemu ayrıca 'nographic' argümanı ile komut satiri üzerinden çalıştırılabildiğinden herhangi bir geliştirme ortamına entegre edilebilme olanağı bulunmaktadır. Qemu sanal makinesi ile sanal bir optik sürücü ile önyükleme yapılabilmesi için sürücü imajının Iso uzantısına ve El Torito[22] formatına sahip olması gerekmektedir. EFI uygulamasına ait dosya yapısı değişmeyecektir. Belirtilen sekile optik sürücü imajı yaratabilecek birçok açık kaynak ve ticari uygulama bulunmaktadır.

5.2.2. Multirust

Multirust[23] sistem üzerine birden fazla Rust geliştirme araçları surumu kurulmasını mümkün kılan bir araçtır. Herhangi bir klasörde hangi surumun veya yayın kanalının kullanacağını belirlemek mümkündür. Ayrıca Rust kurulumunu güncellemek veya kaldırmak Multirust özellikleri arasındadır. İşletim sistemi bileşenleri Nightly yayın

[19] http://wiki.qemu.org
[20] http://www.tianocore.org/edk2
[21] http://www.linux-kvm.org/page/OVMF
[22] https://support.microsoft.com/en-us/kb/167685
[23] https://github.com/brson/multirust

kanalını kullandığından diğer üst seviye uygulamalar ile birlikte aynı sistem üzerinde geliştirildiklerinde Rust geliştirme araçlarının sürekli olarak silinip kurulması problemini ortadan kaldırmaktadır. Ayrıca Nightly doğası gereği deneysel özellikler içerdiğinden sürümler arası hızlıca geçişe olanak vermesi oldukça faydalıdır.

Tablo.8. Temel Multirust komutları

Multirust komutu	Olası parametreler	Klasör için görevi
Override	Nightly, Beta, Stable	Kullanılacak Toolchain belirlenir
Show-override	-	Var olan Override gösterilir
Update	Nightly, Beta, Stable veya argüman olmadan hepsi	Belirtilen Toolchain güncellenir
Remove-override	-	Var olan Override silinir

5.3. Örnekler

C dili 70li yıllarda tasarlandığında olduğu gibi yakın gelecekte de bilgisayar mimarileri Program Counter, Register ve adreslenebilir hafıza ile çalışmaya devam edecek. C soyut makine modeli birçok donanım platformuna oldukça uyumlu olsa da tür sistemi aşırı bir şekilde eskidir. Rust aynı soyut makine modelini korur fakat dil ara yüzü olarak oldukça ileridedir [28].

```
1  #[no_mangle]
2  pub extern "win64" fn efi_main(
3      _image: *const EFI_HANDLE,
4      system_table: *const EFI_SYSTEM_TABLE) {
5          let sys = unsafe { &*system_table };
6          let bs = unsafe { &*sys.BootServices };
7
```

Örnek bir önyükleyici ilk olarak Boot Services tablosunu almalı ve ekrana sistem durumu hakkında çıktılar yazdırarak başlamalıdır. EFI API ara yüzü C dili için tasarlandığından Rust veri yapıları hafızada saklandıkları model açısından C diline benzer şekilde olmalıdırlar.

```
#[repr(C)]
pub struct EFI_TABLE_HEADER {
    pub Signature: u64,
    pub Revision: u32,
    pub HeaderSize: u32,
    pub CRC32: u32,
    pub Reserved: u32,
}
pub struct EFI_HANDLE;

#[repr(C)]
pub struct EFI_SYSTEM_TABLE {
    pub Hdr: EFI_TABLE_HEADER,
    pub FirmwareVendor: *const u16,
    pub FirmwareRevision: u32,
    pub ConsoleInHandle: *const EFI_HANDLE,
    pub ConIn: *const (),
    pub ConsoleOutHandle: *const EFI_HANDLE,
    pub ConOut: *const EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL,
    pub ConsoleErrorHandle: *const EFI_HANDLE,
    pub StdErr: *const (),
    pub RuntimeServices: *const (),
    pub BootServices: *const EFI_BOOT_SERVICES,
    pub NumberOfTableEntries: u64,
    pub ConfigurationTable: *const (),
}
```

EFI yazı dizileri 16-bit UTF karakterlerden oluşur. Rust dilinde Char türü ise 32-bit karakterlerdir ve Encoding yapıları farklıdır. Cons sınıfı ekrana çıktı yazdırmak için kullanılacak ConsLf ise konsola satir atlama ve satir başı uygulayacaktır.

```
1  pub struct Cons<'a>(&'a EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL);
2  pub struct ConsLf;
```

'core:: fmt:: Display' Trait Write!, Format! veya Println! gibi formatlanmış yazılar içerisinde '{}' karakteri ile uygun alana yerleştirilirken nasıl yazıya çevrileceğini tanımlar.

```
1  impl Display for ConsLf {
2      fn fmt(&self, f: &mut Formatter) -> fmt::Result {
3          write!(f, "\r\n")
4      }
```

Tüm ilkel Rust türleri için bu Trait standart kütüphane tarafından tanımlanmıştır.

```
1  pub type EFI_TEXT_RESET = *const ();
2  pub type EFI_TEXT_STRING = extern "win64"
3      fn(*const EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL, *const u16) -> u64;
4
5  #[repr(C)]
6  pub struct EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {
7      pub Reset: EFI_TEXT_RESET,
8      pub OutputString: EFI_TEXT_STRING,
9  }
```

Ardından Write fonksiyonu Rust Char türü için iki ve Null satir sonu karakteri için 48-bit boyutundaki tampon bellek ile UTF-16 cevrimi yaparak EFI ekran konsoluna yazdırma işlemini gerçekleştirir.

```
1  pub fn write(&self, string: &str) {
2      let out = self.0.OutputString;
3
4      for i in string.chars() {
5      let mut s = [0 as u16; 3];
6      if i.encode_utf16(&mut s).is_none() {
7          s[0] = '?' as u16;
8          }
9      out(self.0, s.as_ptr());
10     }
11 }
```

Boot Services tablosu ile gelen Vendor alanı EFI konsol formatına uygun olduğundan direkt olarak ekrana basılabilir. Merhaba mesajı ise Cons sınıfı aracılığı ile yazdırılmaktadır.

```
8  // efi_main devam
9  let output = unsafe { (*conout).OutputString };
10 let exitboot = unsafe { (*boot).ExitBootServices };
11
12 output(conout, vendor);
13
14 let mut cons = unsafe { Cons::new(&*conout) };
15 write!(&mut cons, "{}", ConsLf).unwrap();
16 write!(&mut cons, "Merhaba Rust !{}", ConsLf).unwrap();
```

Benzer şekilde sadece konsol çıktısı için değil örneğin dosya işlemleri için de Rust ve EFI yazı türleri arasında cevrim yapmak gerekebilmektedir. Aşağıda gösterilmekte olan EfiString sınıfı 50 karakterlik UTF-16 tampon belleği kullanarak Rust ve EFI yazı türleri arasında çevrim yapmak ve sonucu saklamaktadır.

```
1  pub struct EfiString {
2      pub buf: [u16; 50],
3      pub len: usize,
4  }
5
6  impl EfiString {
7      pub fn from_str(s: &str) -> Self {
8          let mut b = [0 as u16; 50];
9          let mut l = 0;
10
11         for (d, c) in b.iter_mut().zip(s.chars()) {
12             *d = c as u16;
13             l += 1;
14         }
15
16         EfiString { buf: b, len: l }
17     }
```

Konsol çıktısının tersine burada yapılmakta olan cevrim direkt olarak türlerin birbirine cevrimi seklindedir. Geçersiz karakterler kontrol edilmemektedirler.

6. SONUÇ

Bu tez çalışmasında Rust programlama dilinin gelişmiş semantikleri ve EFI önyükleyici aygıt yazılımının temel noktaları anlatılmaya çalışılmıştır. Rust diline ait önyükleyici tasarımında kullanılabilecek dil özellikleri tartışılmış, bu alanda kullanılan ve en yaygın diller olan C ve C++ dilleri ile karşılaştırmalar yapılmakla birlikte bu alanda yapılan çalışmalardan bir kısmına ait fikir ve düşünceler de paylaşılmaya çalışılmıştır.

Rust programlama dili sadece önyükleyici tasarımına değil sistem programlamayı tekrardan kara tahtaya yatırmakta ve tabu olarak kabul edilmiş bu alandaki sorunları kökten ve kesin bir şekilde çözmeyi amaçlamaktadır. Ayrıca üst seviye başka dillerden yapısına kattığı birçok paradigma sayesinde C dilinden bu yana yasanmış birçok gelişmeyi Rust dili ile alt seviye tasarımlar yapılırken kullanmak mümkündür. Sadece dilin kendisi değil oldukça esnek olan derleyicisi, paket yöneticisi ve etrafındaki sağlam organizasyon topluluğu sayesinde Rust diğer sistem programlama dillerinin aksine vahşi bati temalı sistem programlamaya son vermektedir.

EFI aygıt yazılımı ise geri uyumluluk ve farklı donanım üreticileri arasında yaşanan standardizasyon problemini ortadan kaldırması açısından devrimseldir. Artık işletim sistemi olmayan cihazlarda ağ üzerinden sistem kurulumu mümkün kılınmış aynı zamanda kullanıcı dostu bir ara yüz ile bunun yapılması sağlanmıştır.

Rust dilinin C dili ile sağlam etkileşimi dile olan ilgi ve geçişi ciddi ölçüde hızlandırmaktadır. Rust şuan için C dili ile iç içe kullanıldığı taktirde bile ciddi oranda hafıza güvenliği artışı sağlamaktadır. Bilgisayar mimarilerinde bugün hafıza güvensizliğinin yarattığı zarar oldukça fazladır. Rust teknolojisi Evan Fuller, Jeffrey M. Rabin, Guershon Harel tarafından yazılan ‘Intellectual Need’ yayınında anlatıldığı üzere öğrencilere öğretilmeye çalışılan matematik dersi gibidir ve dile olan ihtiyacın anlaşılmasıyla yayılmaya devam edecektir. Fakat bu ihtiyaç ekonomik veya sosyal değil entelektüel bir ihtiyaçtır [23].

KAYNAKLAR

[1] Balbaert, I., *Rust Essentials*, s. 1-16, s. 89-93, s. 118-119, Packt Publishing, 2015.

[2] Zimmer, V., Rothman M., Marisetty S., *Beyond BIOS: Developing with the Unified Extensible Firmware Interface 2nd Edition,* s. 1-19, Intel Press, 2011.

[3] Petzold, C., *Code: The Hidden Language of Computer Hardware and Software*, s. 40-47, Microsoft Press, 2000.

[4] Okasaki, C., *Purely Functional Data Structures*, s. 1-7, Cambridge University Press, 1999.

[5] Avram, A., *Interview on Rust, a Systems Programming Language Developed by Mozilla,* http://www.infoq.com/news/2012/08/Interview-Rust, 01.03.2016.

[6] Metz, C., *The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire,* http://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire, 01.03.2016.

[7] Rust official book, *References and Borrowing*, https://doc.rust-lang.org/book/references-and-borrowing.html, 01.03.2016.

[8] Turon, A., *Stanford Seminar,* https://www.youtube.com/watch?v=O5vzLKg7y-k, 01.03.2016.

[9] Turon, A., *Abstraction without overhead: traits in Rust,* http://blog.rust-lang.org/2015/05/11/traits.html, 01.03.2016.

[10] Turon, A., *Fearless Concurrency with Rust,* http://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html, 01.03.2016.

[11] Kidd, E., *Bare Metal Rust 2: Retarget your compiler so interrupts are not evil*, http://www.randomhacks.net/2015/11/11/bare-metal-rust-custom-target-kernel-space, 01.03.2016.

[12] Yao, J., Zimmer V., *A Tour beyond BIOS Memory Map Design in UEFI BIOS,* https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Memory_Map_in%20UEFI_BIOS.pdf, 01.03.2016.

[13] Høiby H., Lefsaker S., *RustyGecko - Developing Rust on Bare-Metal*, Norwegian University of Science and Technology, 2015.

[14] Beingessner, A., *You Can't Spell Trust Without Rust,* Carleton University, 2015.

[15] Light, A., *Reenix: Implementing a Unix-Like Operating System in Rust,* Brown University, 2015.

[16] Levy A., Andersen M., Campbell B.,, Culler D., Dutta P., Ghena B., Levis P., Pannuto P., "Ownership is Theft: Experiences Building an Embedded OS in Rust", *Proceedings of the 8th Workshop on Programming Languages and Operating Systems, PLOS 2015,* pp. 21-26, 2015.

[17] Rust official book, *Release Channels,* https://doc.rust-lang.org/book/release-channels.html, 01.03.2016.

[18] Munksgaard P., Jespersen T.B.L., "Practical Session Types in Rust", *WGP 2015 Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming,* pp. 13-22, 2015.

[19] Levis P., Madden S., Polastre J., Szewczyk R., Whitehouse K., Woo A., Gay D., Hill J., Welsh M., Brewer E., Culler D., "TinyOS: An Operating System for Sensor Networks", *Ambient Intelligence,* Part II, pp. 115-148, 2004.

[20] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide,* http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf, 01.03.2016

[21] Patton, R., *Software Testing*, Sams Publishing, s. 9-21, 2001.

[22] Preston-Werner, T., *Tom's Obvious, Minimal Language*, https://github.com/toml-lang/toml, 01.03.2016

[23] Fuller E., Rabin J.M., Harel G., *Intellectual Need and Problem-Free Activity in the Mathematics Classroom*, http://math.ucsd.edu/~jrabin/publications/ProblemFreeActivity.pdf, 01.03.2016

[24] Rust FAQ, *Frequently Asked Questions*, https://www.rust-lang.org/faq.html#what-is-monomorphisation, 01.03.2016

[25] Hobden A., Coady Y., *Understanding Over Guesswork*, https://hoverbear.github.io/rust-education-paper/paper.pdf, 01.03.2016

[26] Rust Nomicon book, *How Safe and Unsafe Interact*, https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html, 01.03.2016

[27] Getreu J., *Enhance Embedded System Security With Rust, Revision 1.1*, https://getreu.net/public/downloads/doc/Enhance%20Embedded%20System%20Security%20With%20Rust/Enhance%20Embedded%20System%20Security%20With%20Rust-Example%20of%20Heartbleed.pdf, 01.03.2016

[28] Poss, R., *Rust for functional programmers*, http://science.raphael.poss.name/rust-for-functional-programmers.pdf, 01.03.2016

[29] Rust official book, *Foreign Function Interface,* https://doc.rust-lang.org/book/ffi.html, 01.03.2016

[30] Fog, A., *Calling conventions for different C++ compilers and operating systems,* http://www.agner.org/optimize/calling_conventions.pdf, 01.03.2016

[31] Clang 3.9 documentation, *Cross-compilation using Clang*, http://clang.llvm.org/docs/CrossCompilation.html, 01.03.2016

[32] Bulusu, M., Zimmer V., *UEFI Plugfest 2015 - Challenges for UEFI and the Cloud,* https://firmware.intel.com/sites/default/files/resources/UEFI_Plugfest_2015_Challenges_in_the_Cloud_Whitepaper_0.pdf, 01.03.2016

[33] Hahn, K., *Robust Static Analysis of Portable Executable Malware,* HTWK Leipzig, 2014.

ÖZGEÇMİŞ

1990 yılında İstanbul ilinde doğdu. Lisans eğitimini tamamladıktan sonra Beykent Üniversitesi Bilgisayar Mühendisliği anabilim dalında yüksek lisans eğitimine başladı. Çeşitli özel kurumlarda yazılım takım liderliği görevini üstlendi. Yabancı dili ingilizcedir.