Utilizing Rust Programming Language for EFI-Based Bootloader Design

Tunç Uzlu and Ediz Şaykol

Beykent University, Department of Computer Engineering,
Ayazağa, 34396, İstanbul, Turkey
tuncuzlu9@gmail.com; ediz.saykol@beykent.edu.tr

Abstract

Rust, as being a systems programming language, offers memory safety with zero cost and without any runtime penalty unlike other languages like C, C++ or Cyclone. System programming languages are mainly used for low level tasks such as design of operating system components, web browsers, game engines and time critical missions like signal processing. Main disadvantages of the existing systems languages are being memory unsafe and having low level design. On the other hand, Rust offers high level language semantics, advanced standard library with modern skill set including most of the features and functional elements of widely-used programming languages. Moreover, Rust can be used as a scripting language like Python, and a functional language like Haskell or any other low level procedural language like C or C++, since Rust is both imperative and functional having no garbage collector. These design choices make Rust a suitable match for low level tasks via including high level scalability and maintainability. Meanwhile, EFI (Extensible Firmware Interface) specification is aimed to remove the limitations of legacy hardware. Hence, we present our analysis of utilizing Rust language on EFI-based bootloader design for x86 architecture, to make it useful for both practitioners and technology developers.

1 Introduction

Rust programming language has been designed by Graydon Hoare and currently it is actively being developed by Mozilla Foundation. It is also being used in Servo, Mozilla Foundations massively parallel web browsing engine, which is unique because of its concurrent process rendering and compositing steps [JML15]. Rust, as being a systems programming language, has ability to operate at the lowest level without any runtime penalty, like C, C++ or Cyclone, but offers complete memory safety, unlike these languages. Systems programming languages are crucial for time criticial tasks like signal processing and also for bare-metal operations such as design of operating system components, web browsers, game engines where raw hardware access is a must. Existing systems languages are memory unsafe and extremely complicated because of their low level nature.

Systems programming languages are considered essential for embedded systems because of low memory availability and exiguous processing power [HL15]. The main reason is the lack of garbage collector which causes non-deterministic delays [LAC+15]. Garbage collectors provide very safe memory management, but poorly manages the memory space and unpredictably runs at the background. This design choice also affects energy consumption which is very important for embedded systems and changes operating system design paradigm [LMP+05].

On the other hand, Rust is both imperative and functional language. Although including different flavors, Rust is highly scalable with capable standard library comparable to high level languages. Rich language semantics and haveing no garbage collector makes Rust suitable match for low level tasks while having high maintainability level. Moreover, Rust can be used as a scripting language like Python or as a functional language like Haskell because of its inherited skill set has been mostly adpoted from modern languages.

C++ is the most powerful systems programming language today. Because of its multi paradigm design and zero cost runtime performance, it is widely

used by numerous organizations and people with different backgrounds. C++ has features with complicated runtime support like RTTI and exceptions disabled for most bootloader applications. As it includes every element from its predecessor C language, it also includes every memory safety pitfall from C. This variation makes C++ even more vulnerable to memory unsafety especially architects with C background widely rely on these language elements. Cyclone, on the other hand, developed as an extension to C language to provide Rust-like memory safety mechanism with ability to port from C to Cyclone without much effort. However, this design choice caused the language semantics to become restrictive and unwieldy.

Another language which is popular and somehow racing with Rust is Go language because of its low learning curve. Go is supported by Google and is a high level language which can be compared to Python or Ruby. Go neither have generic types nor provides safety over its concurrency model, Goroutines. Rust has generics with monomorphisation so they are statically dispatched and has good runtime performance [Bal15].

Here, we present our analysis of utilizing Rust language on EFI-based bootloader design for x86 architecture, to make it useful for both practitioners and technology developers. Our analysis in this paper starts with presenting Rust language basics in detail in Section 2. Then, bootloading basics is presented in Section 3. Since the main idea behind using Rust is programming a critical-and-safe low-level task with high-level programming concepts, we found bootloader design a typical application for this purpose, and discuss design choices that make Rust suitable in Section 4. Finally, Section 5 concludes our paper and states future work.

2 Rust Language Details

Rust is an open source programming language, including an issue system for bug reporting and separate RFC tracker for language standardization, which are located on Github repository. With the help of numerous contributors around the world, Rust provides precompiled development environment for Linux, Windows and OS X. It is also possible to cross compile Rust for Ios, Android, Rasperry Pi and other operating systems. As Rust is a separate development toolchain from operating system, it is radically closer to deterministic code generation process. Hence, Rust is completely decoupled in this perspective. On the other hand, languages like C or C++ depends on header files and libraries through the operating system, lots of applications along with various operating system distributions and updates might influence the collection.

Rust ecosystem includes *Rustc* compiler but also a very powerful package manager, *Cargo* with its registry webpage for crates, *Rustfmt* for code formatting, and *Rustdoc*. for automatic document generation. Cargo has very well dependency management as it offers strict versions of dependencies to be defined. It allows arbitrary flags to pass to Rustc, the Rust compiler, but most importantly with target argument [HL15] it is possible to cross compile to another system differentiating from host operating system. There is also features argument for conditional compiling. Cargo reads projects meta information from a Toml file which is very much like JSON, but more suitable for human editing, rather than data serialization.

2.1 Rust Programming Concepts

Ownership is one of the most important language semantics of Rust. Variable bindings can have one unique owner. They can be moved, can be borrowed numerous times if they are not previously borrowed as mutable, that can be happened only once. Ownership also works on resources like files or sockets and across threads. Rust provides traits to offer functionality similar to inheritance [JML15]. For example, to duplicate an object Rust have Clone trait [LAC+15] also there is Copy trait for bitwise copying. Anonymous closure functions are also defined in terms of traits in Rust like *Fn* or *FnMut* depending on mutability and if the closure is called once it should be *FnOnce*. They can not be used as a return value so they should be enclosed into a Box which allocates space from Heap memory [Lig15].

Rust have Structs in a very similar way to C. The main difference is data structure itself may be public whereas its elements may be private in the code space. Rust offers algebraic Enum which is more functional and much more advanced compared to that of C++, which only has type checking. Option generic type is a special Enum type with maybe characteristic. It is being used as a selector between a return value, Some, or an error value, Err (or absence None). This Option and Error types are suitable for representing Null pointers so that it is impossible Rust to have Null pointer errors. This paradigm is also suitable for Null pointer optimization as Rust uses LLVM compiler infrastructure and benefits from same backend optimizations of C language family. Pointer safety is guaranteed with holding *Lifetimes*. Like type inference, reference lifetimes can be guessed by Rust compiled and this is called lifetime elision. Sometimes explicit lifetime marks are required as references lifetime must be equal or larger than its originating binding.

Concurrency is the core of Rust. Same owner-

ship mechanism applies across threads and Rust offers thread safety mostly on compile time. Channel, for example, allows data to be send safely across threads if the type satisfy Send Marker trait. Markers are Rusts internals to enforce safety rules. Other important markers are Sync, can be shared across threads, Sized, type has a known size at compile time. When multiple threads need to modify same region of memory classical lock mechanisms like Mutex or RWLock are provided. The key point is locking in Rust works on the data itself, not on the code. Software architects using C++ tries to prevent data race by locking the code itself by design.

A well-known analysis on the cost of software testing [Pat01] states that if a design error at the specification phase costs about zero to 10 cents, in the software testing phase it costs 1 to 10 dollars. However, if the error is found by the eventual user the cost is at least 100 dollars, hence the increase is logarithmic. To help in reducing the errors, Rust is designed to be a strong and static language. Dynamic languages suffer from compiler aid or lack of typing depending on language design. They have low learning curve and high portability or embedibility. On the other hand, languages with strong typing such as Rust or Haskell have higher learning curve but provide superior type safety at compiling stage. Compilers are far better at catching bugs than human eye. There are also weak static languages exist. They offer automatic type conversion and this unpredictability causes bugs just like dynamic languages. Undefined behaviors have always been spots for hard to find bugs. For example, C++ language, unlike Rust, does not define size of its main integer type, int, or char type can be signed or unsigned depending on various factors like compiler, operating system or building flags.

Charles Petzold described a telegraph relay as a device that a clicker and a sound magnet connected with a stick by lazy operator. Because they were moving simultaneously [Pet00]. As it is acceptable for the operator to make mistakes when hearing the Morse code for a day and clicking the correct dash or dot code as there is no mechanical aid. Dynamic languages are somehow the same. Compiler support is an example for the relay device, with strong type checking, is seriously important to prevent human errors. Rust takes this a step forward by providing compile time memory and thread safety. Runtime checks are done only if there is no any other choice, like bound checking for arrays.

Rust also have borrowed functional elements from various languages, for example, Iterators. They are lazily evaluated and offers numbers of higher order functions when an iterator is defined or converted into. Functional flavor is harder for systems programming audience. Like borrowing a master chefs knife, imperative paradigm is powerful when used correctly, but tend to fail because of its destructive nature on global data [Oka99].

2.2 Comparing Rust with C and C++

Rust is the remedy for numerous systems programming bugs by design. First one is buffer overflow or underflow on arrays. C++ has no bounds checking for arrays so writing or reading outside of bounds may cause corruption or page fault depending on operation. Rust checks array bounds at runtime because there is no way to detect array size at compile time. Also Rust does not allow indexing operation with negative argument. Array elements are accessed with Index trait and this trait is not defined for negative values. At last integer overflow remains. Fortunately, Rust checks for arithmetic overflows if the number is unsigned. This type of corruption is the main source of buffer related attacks for years.

The second is iterator invalidation. With C++, while an iterator is looping over a collection and the collection has been modified, this causes the iterator to be invalid. Data is corrupt or iterator goes into an infinite loop depending on operation. With Rust, as the collection is borrowed by the iterator, it can not be borrowed mutably by modifier functions like Push [Bei15].

The last one is use-after-free memory bugs. High level languages prevent this kind of error by using garbage collector while Rust has its unique ownership and lifetime semantics to prevent this memory pitfall with zero runtime performance cost. Rust also has hygienic macros and the macros are part of AST transformation [Lig15].

Rust has unsafe blocks for non-ideal conditions like dereferencing raw pointers, type transmute or foreign function interface. With Rust, there is no possibility to cause concurrency failure outside of unsafe block even if the design of application is tremendously bad. Raw pointers are ideal for storing MMIO or interrupt controller, system tables memory address as they are stored on constant memory location. C language does not prevent pointers to be modified outside of their lifetime this is a problem with Rust only when unsafe is used. Rust also offers strong foreign function interface to C language with Extern keyword and talking to C has no runtime performance cost. This makes calling foreign function from EFI is extremely simple with a simple binding module.

3 Bootloading Basics

3.1 Legacy Bootloading

Bootloaders are responsible for building memory map, finding system tables and launching operating system kernel. For backwards compatibility reasons CPUs with x86 architecture used to start in 16-bit real mode which only has access to 1MB of memory. Typical routine of a bootloader should be first enabling higher memory over A20 gate [Cor16]. Bootloading concepts heavily relies on chipset specification and BIOS interrupts. As they are designed by different hardware vendors, conflicts exist on different systems. Such units have grown organically over years and they have poorly standardized.

Next step should be enabling protected mode, which provides 32-bit addressing and paging. Activation of paging is mandatory and also very useful as it provides separation between kernels and user applications pages in terms of permissions. Also paging is the key for virtual memory along with creation noexecutable pages to prevent runtime code execution from text sections. Paging is also being used on high level, for example guard paging is being used to grow stack when there is a page fault exception at the end of program stack. On real mode there is another memory management called segmentation. It works by using different selectors for sectioning areas of code and data blocks. After protected mode switch segmentation is now obsolete, but at the same time it is still active and has to be configured such as it should provide the same flat addressing. Some segment registers are still being used in Linux kernel to detect buffer overflow over function call return address on stack.

Lastly, there is long mode with provides 64-bit addressing in canonical form and removes historical features like BCD [Cor16]. Different kernels have strict requirements about the state that it is going to be started. There are also various sub-modes like for emulating real mode interrupts in protected mode, called virtual-8086 mode, or emulating complicated driver-required devices in early modes, called system management mode. Between this mode switches interrupt controller must be reconfigured correctly. At the old times real mode interrupts which were invoking appropriate BIOS support were being used in place of device drivers in order to talk to the hardware.

As devices became much more complicated operating systems took over all hardware interaction. BIOS were started to be used as a bootloader firmware. Its complex nature was such a boredom and also lack interaction with modern technology, such as network access, was led Intel to design EFI specification which is a modern platform firmware for bootloading. EFI can run applications just like an operating system and most importantly runs the system in long mode.

3.2 Unified Extensible Firmware Interface (UEFI)

EFI specification has been designed by Intel in 1999 and now it maintained by UEFI consortium that includes more than 160 companies [ZRM11]. EFI has lots of modern features such as networking, human interface device support and bootloader driver model. It provides safer way to update firmware update with packages, Capsules, that enforce EEPROM validation [BZ15]. The flowchart of EFI-based bootloading process is shown in Figure 1.

EFI is built up with numerous modules while boot, runtime and driver modules are mandatory. Boot module is the key to generating memory map and locating systems tables. x86 memory model, while depending on memory controller or chipset, has lots of gaps in the memory [YZ15]. These include MMIO, configuration registers for PCI devices4, legacy timers, video frame buffers or regions belongs to ACPI or interrupt controller tables (reclaimable or not). As brute-forcing to generate a memory map is extremely unstable, EFI provides the map out of the box. Driver model allows to create drivers for file systems or NIC devices for richer bootloading environment. While runtime module offers monotonic timers, system time, power supply commands or firmware updating.

EFI bootloader applications can be developed with Rust like any other applications uses foreign function interface, but there should be no standard library for all types of operating systems. The library of Rust is rich as high level languages. Most of the language characteristics provided over standard library and not embedded into languages itself. Rust binaries should be linked into a final Portable Executable (PE). PE file format is being used in Windows operating system and offers sectioning along with relocation [Hah14].

4 Designing EFI-based Bootloader with Rust

In order to create an EFI application with Rust, first *Libcore* should be compiled for target platform. Libcore is the bare-metal subset of Rust standard library that has no operating system dependency. A few memory functions are needed to build Libcore, which can be obtained from *Rlibc*. It is also possible to use their C counterparts. EFI application, Rlibc library and Libcore should be cross-compiled to target system by correct triplet. Although x86_64-pc-windowsgnu is the most suitable triplet (because of a future PE linkage) for such a bootloader application, it is not sufficient.

There should be a custom target triplet definition file in JSON format and it should disable few language

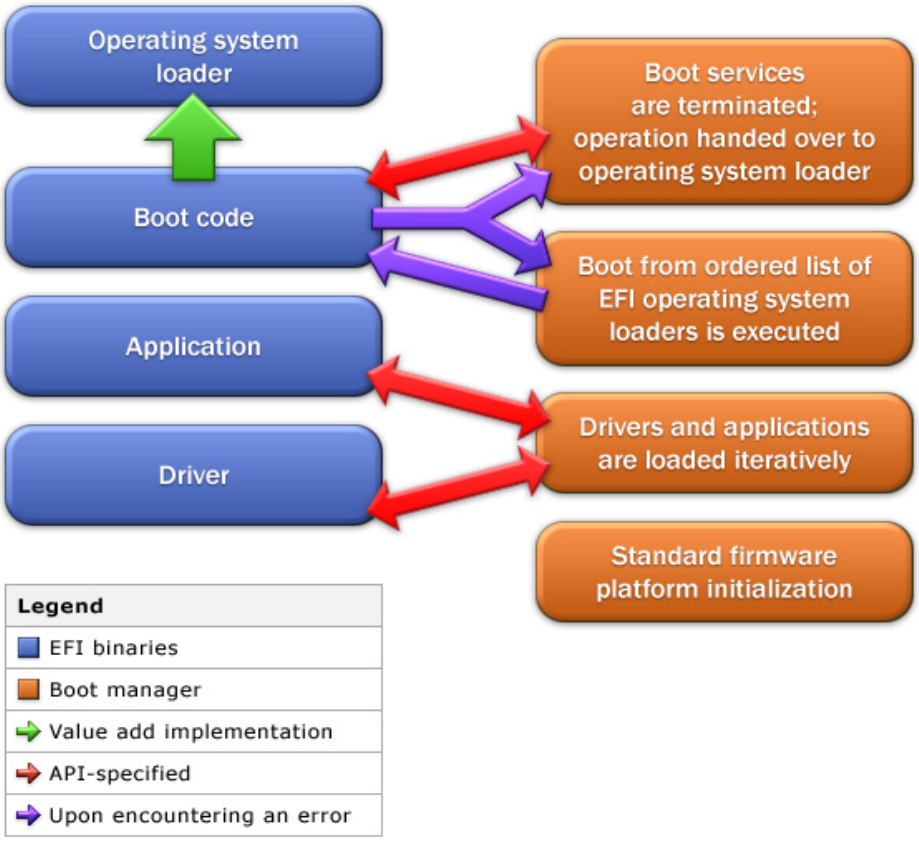

Figure 1: The flowchart of EFI (Source: https://en.wikipedia.org/wiki/Uni-fied_Extensible_Firmware_Interface).

features.

- First of them is *Compiler-rt*, because otherwise LLVM compiler infrastructures helper library or Rust languages itself should be reconfigured and recompiled for target architecture even though there is no need.

- Second one is *Morestack*, as there is no highlevel memory management Morestack is not declared by the application and stack is managed manually so compiler should not define Morestack.

- Third one is stack unwinding as when an exception occurs in a bootloader, there is little to no chance to recover. It is also known as landing pads in Rust and can also be defined as compiler flag.

- Finally, floating point operations and optimizations must be disabled from the triplet configuration file. It has been found that floating point optimizations corrupts interrupt handlers with bare-metal Rust [HL15]. Also in bootloader environment, floating point stack or coprocessor have not yet configured. Also most operating system kernels does not provide floating point functionality in kernel space. Along with the FPU stack and SSE, there are also other mathematical floating point units such as MMX and 3dNow depending on CPU model. LLVM does not allow us to disable floating point support in such state because Libcore library has floating point code. It should be modified and cleaned from floating point in order to be used in kernel or bootloader programming. One example can be that Fxsave or Fxstor instructions copy every FPU storage registers into stack between function calls.

The EFI application then can be linked with subsystem 10 flag, put into FAT32 drive and tested with a computer or virtual machine. *Ovmf* is an open source BIOS for *Qemu* having EFI support. Qemus nographic option makes it easy to integrate into any development environment. There is also a tool called *Multirust* which crates Rust version overrides for folders. It makes easier to make switch between nightly versions or stable release of Rust. EFI also has a shell which is a helper for bootloader design. For example, Pci command lists pci device paths or Memmap shows the memory map. EFI Capsules also support I2C which can be used to flash ROMs belonging other hardware.

Historically bootloaders consisted two or three phases. They were loaded into memory step by step, upgraded the system to a higher mode and prepared

the environment for the next phase. This is no longer required with EFI, but it is possible to keep this design. As an EFI application relies on its own binary structure and calling convention, it may beneficial to use a second stage bootloader which has been started from EFI. This second stage application is not subjected to EFI specification and is just a small kernel indented to run the real kernel.

There are numerous resources on operating systems design with Rust including [HL15] and [Lig15]. All resources with C language are applicable to Rust since the syntactic elements of these two languages are similar. Also Rusts strong foreign function interfaces provides strong interaction. C is lingua franca of systems languages. It has very good runtime performance and has raw memory management capability. Its abstract machine model perfectly fits into current hardware which utilizes program counter, registers and addressable memory, but its type system has aged [Pos14]. Rust, on the other hand, is fresh and brings lots of modern features from newer high level designs. It offers safety at compile time and abstractions are zero-cost at runtime.

5 Conclusion and Future Work

In this paper, the advanced semantics of Rust programming language is presented to clarify the possible use within EFI-based bootloader design process. Various design alternatives and choices are mentioned and the point that make Rust a better choice are discussed. Since one of the main ideas behind using Rust is programming a critical-and-safe low-level task with high-level programming concepts, we found bootloader design a typical application for this purpose

As discussed, Rust offers high level language semantics, advanced standard library with modern skill set including most of the features and functional elements of widely-used programming languages. Moreover, Rust can be used as both a scripting language or a functional language. Additionally, it can also be used as a low level procedural language since it is both imperative and functional having no garbage collector. These design choices make Rust a suitable match for low level tasks via including high level scalability and maintainability.

From the bootloading perspective, the future seems to be based on EFI on x86 hardware. It currently allows end users to download operating system from the Internet and install easily. Today memory unsafety causes serious problems, hence adaptation of Rust is not economical or social, it is intellectual. As our future work, we plan to develop a prototype based on this design process and validate the use of Rust via performance experiments.

References

[Bal15] I. Balbaert. *Rust Essentials*. Packt Publishing, May 2015.

[Bei15] A. Beingessner. You can't spell trust without rust. Master's thesis, Charlton University, Department of Computer Science, 2015.

[BZ15] M. Bulusu and V. Zimmer. Challanges for UEFI and the cloud. In *UEFI Plugfest 2015*, May 2015.

[Cor16] Intel Corporation. Intel 64 and IA-32 architectures software developers manual volume 3 (3a, 3b, 3c and 3d): System programming guide. Technical report, Order Number: 325384-058US, April, 2016.

[Hah14] K. Hahn. Robust static analysis of. portable executable malware. Master's thesis, HTWK Leipzig, Department of Computer Science, December 2014.

[HL15] H.W. Hoiby and S. Lefsaker. Rustygecko - developing rust on bare-metal - an experimental embedded software platform. Master's thesis, Norwegian University of Science and Technology, 2015.

[JML15] T.B.L. Jespersen, P. Munksgaard, and K.F. Larsen. Session types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 13–22, New York, NY, USA, 2015. ACM.

[LAC+15] A. Levy, M.P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto. Ownership is theft: Experiences building an embedded os in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS'15, pages 21–26, New York, NY, USA, 2015. ACM.

[Lig15] A. Light. Reenix: Implementing a unix-like operating system in rust. Master's thesis, Brown University, Department of Computer Science, April 2015.

[LMP+05] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. Springer Verlag, 2005.

[Oka99] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[Pat01] R. Patton. *Software Testing*. Sams Publishing, 2001.

[Pet00] C. Petzold. *Code: The Hidden Language of Computer Hardware and Software*. Microsoft Press, 2000.

[Pos14] R. Poss. Rust for functional programmers. http://science.raphael.poss.name/rust-for-functional-programmers.html, July 2014.

[YZ15] J. Yao and V. Zimmer. A tour beyond bios memory map design in UEFI BIOS. Technical report, Intel Corporation, February 2015.

[ZRM11] V. Zimmer, M. Rothman, and S. Marisetty. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface 2nd Edition*. Intel Press, January 2011.