The Rust Programming Language

CS395

Evan Duffield, Katie Haney-Osborn, Dylan Morris

University of Northern Colorado

Abstract

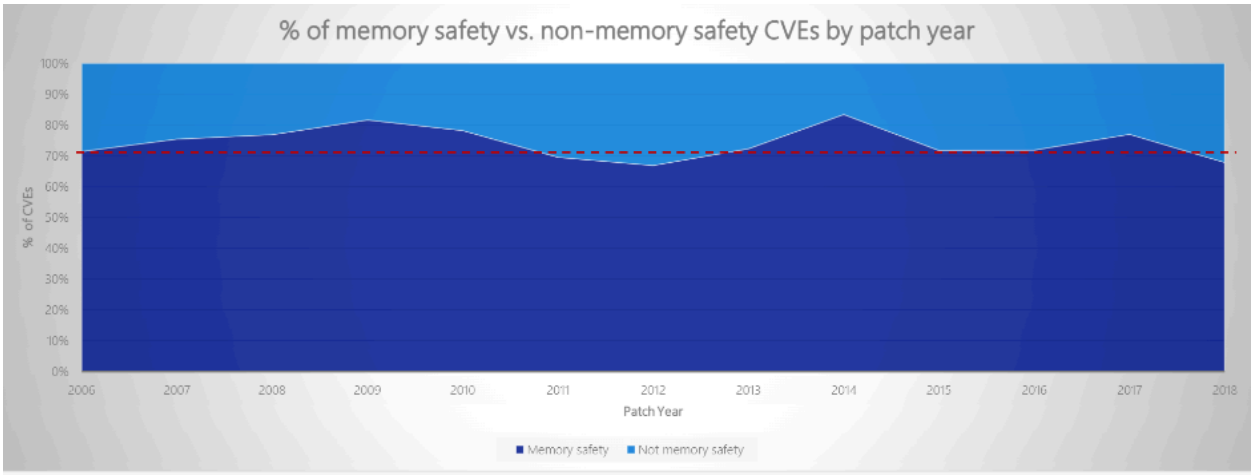
The Rust programming language, while only being in stable release since 2015, has quickly become a serious alternative to the traditional tools used in system programming. Created by a Mozilla employee, Rust was designed from the ground up to deal with the many issues that arise from using C and C++, specifically memory errors. (Jedda) For several years now, the Rust programming language has been ranked as one of the most well loved languages according to the Stack Overflow Developer Survey. (Donovan). This paper will go into detail on why Rust was created and what problems it solves in the world of system programming. Several features of the language will also be explained before an analysis to see if Rust has the ability to replace C/C++ in solving the problems of their domain.

Introduction

Graydon Hoare was a developer for Mozilla in 2006. One day, when he was returning to his apartment, he found that the elevator was out of order due to a software crash. He knew that due to the simplicity of the embedded system, the bug was most likely caused by a memory bug from using C/C++. (Thompson) Hoare felt that it was a serious issue that programmers weren't able to program something as simple as an elevator reliably. This epiphany led to the development of Rust and its key design goal: provide memory safety features without sacrificing efficiency or control for critical systems.

To understand the value of a new language like Rust and why it has succeeded in places where older, more refined languages haven't, the prevalence of memory bugs needs to be explained. At a Microsoft BlueHat Israel conference, the company revealed that for over 20 years about ~70% of security vulnerabilities that have been patched have been due to a issue concerning memory safety. (Miller) Memory safety has become such a focus at Microsoft that the Chief Technical Officer of the Azure platform called to "halt starting any new projects in C/C++ and use Rust for those scenarios where a non-GC language is required." (Claburn). This is one of the reasons Rust is an attractive option for many organizations. The features of Rust make it so its adoption helps cut down on memory bugs, leading to more secure software and less man hours spent on patching existing software. This problem isn't just endemic to large companies but nearly every codebase that utilizes a memory-unsafe language. The Cybersecurity & Infrastructure Security Agency, or CISA, has noted that memory safety has been exploited for

over 50 years, and recommended software manufacturers to use Rust while working to eliminate memory safety vulnerabilities in their codebases. (Lord)



Examples of Memory Safety

What does a memory vulnerability look like in the wild, and how would using Rust solve the bug in question? A great example would be to look at the 1988 Morris worm, one of the most famous and damaging viruses that utilized memory vulnerabilities in software. The worm was created by a graduate student at Cornell who wanted to see how insecure the software used for networking the early internet was. (Bulik) At this point in time most machines that ran the internet used Berkeley Unix, a distributed operating system that was responsible for pioneering networking. Morris exploited the gets() routine from the standard C library, a function that was utilized in the *finger* utility to read user input. The gets() routine generates a limited-sized buffer to read in user data, but the contents can spill into memory if too much data is recieved. Experienced programmers know to either check the size of the input or use other functions, but many still choose to use older calls for simplicity's sake. This was exploited as assembly code with instructions to copy the worm was put at the end of user input. When the data was read, the machine code would be able to hijack the control flow of the *finger* program running on the target machine to spread the worm. (Spafford)

This type of exploit cannot be fixed by "patching" C or C++ as the committees who define these languages expect the users of the language to implement error-checking themselves or through another library. The inherent trust the language designers give the developer is a

benefit to some but leads to the same mistakes being made by every generation of programmers. Jon Erickson, a cryptologist, was able to recreate a demo of a buffer overflow on a completely up to date Linux machine. His machine code, however, contained the linux system calls to fork a new process and to create a root shell. (Erickson) With badly written C/C++ code still having an effect on modern systems, Rust stands out by fixing some of the "issues" that lead to human errors in programming by making it more difficult to have these bugs in compiled programs. For example, Rust fixes buffer overruns by abstracting arrays of data into proper types that have definite bounds in memory. This is in contrast to C/C++, where arrays are really just pointers that store the memory location of the first element. While C/C++ forces programmers to keep track of the sizes of arrays to avoid over indexing or overrunning them, Rust always keeps track of the size of an array so the thread can panic when something is indexed out of bounds. This one change may seem insignificant, but buffer overruns make up around 10-15% of patched vulnerabilities every year. (Blandy)

Rust has other features like its Array typing that guard against vulnerabilities, mostly dealing with issues brought up with pointers. For example, another common vulnerability caused by inexperienced C/C++ developers is null pointer dereferencing. In short, this is when the programmer accidentally tries to use a pointer that has been set to null. This causes the program to try and read an invalid memory location, which could lead to crashes or undefined behavior. This is solved in Rust through eliminating the ability to pass null values entirely, opting instead to create an Option<> type. This type allows programmers to let the program know if a variable could have absent data in a more graceful way in cases such as a function being unsuccessful. Another problem people often experience that Rust solves are dangling pointers. Dangling pointers are similar to null pointers, but instead of pointing to an invalid memory address, they point to data that has been removed from memory. This can happen in several ways, but a common scenario is the programmer using a pointer to data generated in a routine. If the variable isn't properly declared in the routine, it will be marked for deallocation once the pointer is returned and the function is finished. If the programmer tries to access that deallocated data with the pointer they got back, it will crash the program as it will try to read and manipulate invalid data. Rust fixes this with its ownership system, which guards variables from deallocation by assigning it an "owner" or a part of the program that accesses the variable. A variable's first owner is usually the function that creates it, and gains potential owners from the other scopes of

the program that point to the variable. A potential owner gets to borrow ownership when using the variable, and the variable is finally deallocated when there are no potential owners left. This not only makes sure the program never points to deallocated data in memory, eliminating dangling pointers, but it also gives Rust a performance advantage over garbage-collected languages that have to use frequent routines to scan the process for data that can be deallocated. (Blandy)

Rust is not a one stop solution for fixing memory safety issues, as just with C/C++ it is still possible for human error to introduce issues. OneSignal is a company that offers an API to send notifications to a business' customer base, and they use Rust for any work they do involving HTTP. They found that certain data types defined in Rust shouldn't be used during asynchronous processes, as the ownership system will give a variable an immortal owner if an asynchronous call is made but never completed. This caused a memory leak as eventually the number of failed asynchronous functions would grow over time before causing the program to crash. This wasn't prevented by any Rust feature, and they were not stopped or warned by the compiler. (Mara) The Rust designers simply expected you to know not to use those data types for async, which was the exact type of thinking that they hated from the C/C++ designers.

Language Features

One of the major things that sets Rust apart from similar programming languages is moving away from being fully object-oriented. It is not a complete departure as there are many crucial aspects to object-oriented programming, but there are some key differences (Klabnik & Nichols). One of the aspects that Rust chooses to keep is having 'objects' that contain data and behavior. Rust's 'structs' and 'enums' have data, and 'impl' blocks provide methods on 'structs' and 'enums'. While those are not explicitly called objects, they maintain that same functionality. Another aspect of object-oriented programming is that encapsulation hides implementation details. What this means is that the fields within the object, or struct for Rust, are private. Those fields are still updatable, but you need to implement methods to access and update them. In Rust, marking a struct as 'pub' makes it so it's usable by other code and still keeps the private fields

like objects typically have.

```
pub struct AveragedCollection {
    list: Vec<i32>,
    average: f64,
}
```

Listing 17-1: An `AveragedCollection` struct that maintains a list of integers and the average of the items in the collection

The thing that keeps Rust from being object-oriented is the lack of inheritance. Inheritance allows for you to reuse sections of code and use a child type in the same places as the parent type, aka polymorphism. Rust uses 'trait' objects rather than inheritance, which ends up being one of Rust's biggest strengths (Klabnik & Nichols). Traits can allow you to reuse code like with inheritance while also being a lot more specialized. Inheritance is at risk of sharing more code than necessary between components since subclasses shouldn't always share all characteristics of the parent class.

```
pub trait Draw {
    fn draw(&self);
}
```

Listing 17-3: Definition of the `Draw` trait

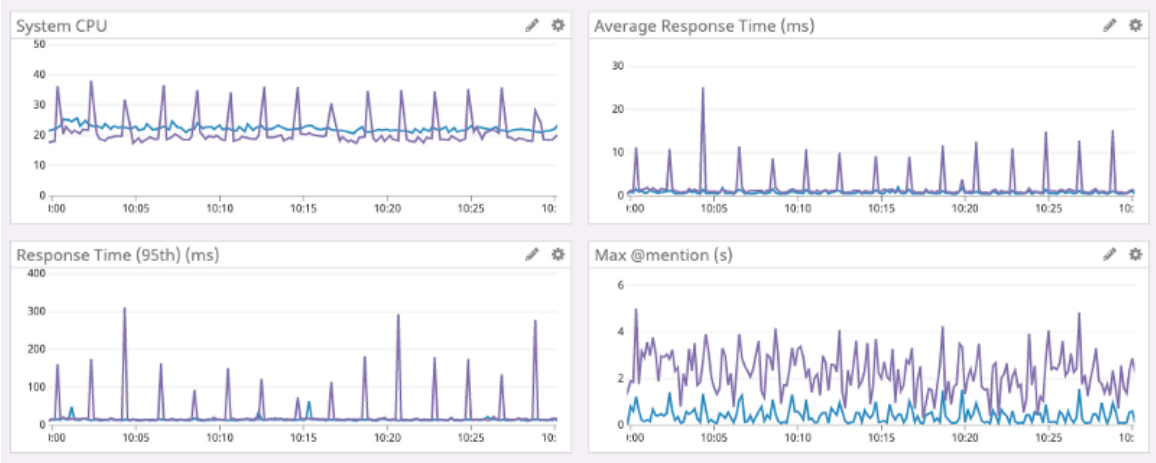
```
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

Listing 17-4: Definition of the `Screen` struct with a `components` field holding a vector of trait objects that implement the `Draw` trait

Rust's lack of full object-oriented programming ends up being its biggest strength compared to other languages. Rust is one of the first major languages to ditch object-oriented programming and more modern languages are taking this approach too. Go is similar to Rust in that it's also not object-oriented. Go has types and methods and allows for object-oriented style programming, but lacks type hierarchy, which makes it identical to subclassing. Go and Rust are both praised for not having object-oriented programming compared to a language like Java.

Java's a widely used language with the classic object-oriented programming structure using its class system. Objects in that language have a state and behavior, along with inheritance.

Rust has several strengths over other languages, like Go. While they both don't have object-oriented programming, Rust additionally has much better garbage collection. In Go, memory is not immediately freed. Instead, garbage collection runs every so often to find any memory that has no references, then frees it. This takes a large amount of time and resources to check if the memory is truly out of use, which slows down the program substantially. Rust has no garbage collection at all and is very memory-efficient and super fast.



Looking at the graph of Discord's response time and system cpu uses (where Rust is blue and Go is purple), there are massive spikes on 2 minute intervals with Go (Howarth). This is entirely because of Go's garbage collection process. When Discord switched to using Rust, those frequent large spikes were entirely reduced and tend to be consistently lower than Go's usage even outside of the garbage collection time periods. When you're using as much memory as a large social media platform like Discord, these inefficiencies really add up. No matter how optimized the code ended up getting, it was impossible to avoid these garbage collection periods that Go has.

Rust also has a phenomenal package manager. Rust uses Cargo since it handles plenty of tasks, such as building code, downloading libraries necessary for code, and building those libraries (Klabnik & Nichols). The following command creates a new directory and project named "hello_cargo" and generates necessary files and a directory of the same name. It also initializes a Git repository automatically, which makes it incredibly useful.

```
$ cargo new hello_cargo
$ cd hello_cargo
```

There's also several useful commands such as "cargo check", which checks if the code can compile, but doesn't produce an executable, making it very useful for debugging (Klabnik & Nichols). There's also "cargo build –release", which compiles the code with optimizations. While it makes the code run faster, it takes longer to compile. Cargo having this along with a debugging option makes Cargo excellent for testing projects.

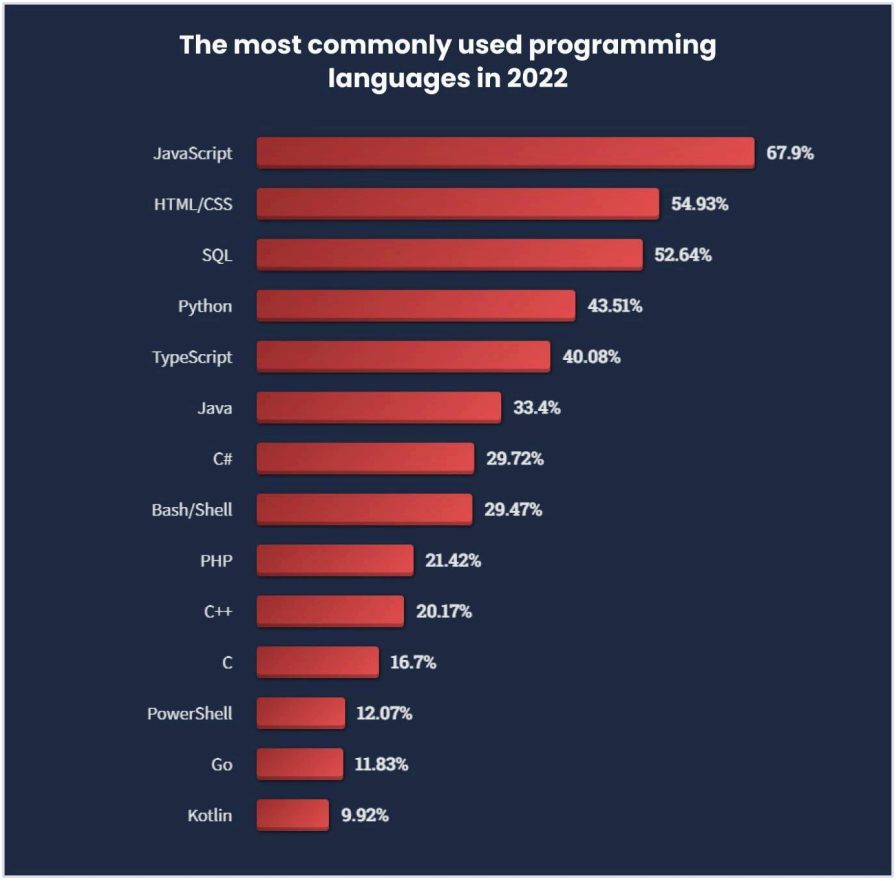
Replacing C and C++

As discussed above, Rust has many advantages over other languages with memory management. Generally, languages either automatically allocate memory and perform garbage collection (IE Python, Java, C#) or require the programmer to take care of these tasks (IE C and C++). Rust innovates this dichotomy with the ownership system, which deallocates memory automatically when the scope that owns the variable ends (Klabnik & Nichols). This avoids the occasional spikes in computer usage that come with occasional garbage collection while preventing the many bugs and frustrating moments that arise from requiring programmers to remember to deallocate memory themselves. Of course, the ownership system can't stop programmers from accidentally using variables in incorrect ways. Fortunately, the Rust compiler works with the ownership system to ensure these bugs don't get past the early stages of development, along with providing helpful tips on how to fix these issues. This is the biggest advantage of using Rust, and the thing Rust was designed from the ground up to be best at. Because of this Rust is an amazing option for any programming task that requires good allocation of memory for any reason, whether it be to maximize performance or just to make sure it can run at all. These factors are especially important for web browsers and operating systems, which both need to run potentially hundreds of tasks at a time while not slowing down the user. That's why Rust's biggest adopters, such as Firefox and Linux, tend to be in these fields (Simone).

Additionally, Rust is a newer programming language made by experienced programmers, beginning in 2006 when the field of programming was already established. This may not seem

like a big advantage, but if you look at when many of the languages in common use today were made, we quickly see that this is not common in the industry. For example, C was developed in 1972, over 50 years ago! Most commonly used programming languages, as seen in this graph, are between 30 and 50 years old. Did we know as much about programming and what we needed from computers 50 years ago as we do today? Certainly not! While these languages have been updated throughout the years, the foundation they're built on is an old one from a completely different era. It makes sense that as the field develops and technology gets better and better that we'd be able to build better programming languages as well, and Rust is one of these better languages. Combined with being in the same niche as C and C++, having literally been designed as a C++ replacement, there's a compelling argument to be made for replacing C and C++ coding with Rust entirely.

However, replacing a programming language is no easy task (Claburn). For other fields it may be easy to replace tools and programs. For example, if you want to switch from using Adobe Illustrator to Krita as an artist, all you need to do is install Krita, learn the UI, download any brushes you need, and you can immediately get back to whatever project you were working on. This is not what programming is like. If you want to switch from programming a web browser in C and instead program in Rust, you have one of two options. First, you can use Rust in a way where it can be incorporated along with the preexisting C code. You won't be able to just stop using C, as you'll need to keep updating that code and make sure it runs properly alongside the Rust code, but it's a far better alternative to the second option, which is to completely reprogram the application in Rust from the ground up. There is no easy transfer option in the way you could open a painting in Illustrator or Krita. Now imagine these options when you're developing Google Chrome or Windows. It's easy to see why we aren't changing these old programming languages, even when alternatives exist that are far better. In fact, most of our commonly used programming languages today have been around for 30 to 50 years (Chekalin)! The graph below shows a survey of programming languages used by professional programmers, most of which are in the age range mentioned before.



In order to see how complex it can be to even implement a new language as an option in an operating system, let's look at how it's being integrated into the Linux Kernel. The project pushing for this integration has already been going for 2 years (Vaughan-Nichols). Additionally, the original branch working to develop it was recently discontinued and replaced with a new branch called rust-next. This is only one of many branches related to the implementation of Rust, which also includes Rust-fixes and Rust-dev. There are also conflicts between the language and kernel themselves, for example deadlocks are permitted in Rust but not in the Linux Kernel, leading to problems where Rust can send invalid code to it. In implementing Rust some unstable features- which are opt-in and subject to change in future versions of the language- were used, which could cause major problems in the future (Rust for Linux maintainers). Additionally, various tools specifically designed for Rust programming are being developed to make integration possible. These consist of versions of tools used in the kernel for C programming repurposed for Rust. As you can see, the process of implementing Rust into Linux isn't easy by any means. It says a lot that all of these Linux developers see Rust as such a worthwhile addition

that it's worth this immense effort, but regardless it's just not possible for this effort to be taken on such a massive scale as to replace C. That would be a far more difficult process.

Just because Rust can't replace C or C++, doesn't mean it can't become a major part of the programming world. As stated before, Firefox and Linux are already using Rust in their projects. Additionally the developers of Android, which is part of the Google corporation, are also considering using Rust in their operating system. If this happens, Rust will be in use by one of the biggest tech companies in the world! And they don't need to replace C or C++ to do so. The incorporation of Rust into preexisting programs will help bring a new standard of memory safety to the world. In addition, Rust can be used to run preexisting C and C++ code, which is how it's primarily being used in Linux right now. So even if we can't replace that old code, we can still improve it by taking advantage of Rust's ownership system to check for flaws. Overall, even if Rust doesn't reach the level of C or C++ for a long time, it still has a bright future in the industry and especially in operating systems development.

Conclusion

Rust has shown itself to be able to create more reliable software that can be used and integrated in professional codebases. Its design allows it to have both the performance and control of languages such as C/C++ with the safety of garbage-collected languages, being a tempting choice for developers who are tired of having to use old tools to solve even older problems. While Rust doesn't make programs perfectly safe like the community tries to claim, the types of vulnerabilities eliminated by its design make up a non-trivial portion of the amount of patching that has to be done for existing software. Rust's modern toolchain and influence from recent trends in language design streamlines the process of writing software compared to its competitors, which may help Rust gain significant traction as newer generations may be resistant to inheriting the technical debt from the C family of languages. Rust most likely will never replace C/C++, as our existing infrastructure is too rooted in legacy codesystems that would require large amounts of time and money to convert to Rust. However, as seen with projects such as Firefox and Linux, Rust is perfectly capable of coexisting with C/C++, and can help by reducing the bugs introduced by never contributions to a codebase.

Works Cited

Blandy, Jim. Why Rust? O'Reilly, 2015.

Bulik, Mark. "1988: 'the Internet' Comes down with a Virus." The New York Times, The New York Times, 6 Aug. 2014, archive.nytimes.com/www.nytimes.com/times-insider/2014/08/06/1988-the-internet-comes-down-with-a-virus/#:~:text=In%201988%2C%20Robert%20T.,5%2C%201988.

Chekalin, Dmytro. "8 Top Programming Languages in 2023." Codica, www.codica.com/blog/top-programming-languages-2023/. Accessed 2 Feb. 2024.

Claburn, Thomas. "In Rust We Trust: Microsoft Azure CTO Shuns C and C++." The Register® - Biting the Hand That Feeds IT, The Register, 20 Sept. 2022, www.theregister.com/2022/09/20/rust_microsoft_c/.

Erickson, Jon. Hacking: The Art of Exploitation. W. Ross MacDonald School Resource Services Library, 2019.

Donovan, Ryan. "Why the Developers Who Use Rust Love It so Much." Stack Overflow, 5 June 2020, stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/.

Howarth, Jesse. "Why Discord is switching from Go to Rust." Discord, 4 February 2020, https://discord.com/blog/why-discord-is-switching-from-go-to-rust. Accessed 22 January 2024.

Jedda, Karim. "Data with Rust." Data With Rust, 21 Jan. 2023, datawithrust.com/chapter_1/chapter_1_2.html.

Klabnik, Steve, and Carol Nichols. The Rust Programming Language, 2nd Edition. No Starch Press, 2023.

Lord, Bob. "The Urgent Need for Memory Safety in Software Products: CISA." Cybersecurity and Infrastructure Security Agency CISA, 1 Feb. 2024, www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products.

Mara, Lily. "Fixing Memory Leaks in Rust." Customer Engagement Blog, Customer Engagement Blog, 23 Nov. 2022, onesignal.com/blog/solving-memory-leaks-in-rust/.

Miller, Matt. "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape." GitHub, https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_

02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf. Accessed 22 January 2024.

Rust for Linux maintainers. “Rust for Linux.” Unstable Features, rust-for-linux.com/unstable-features. Accessed 2 Feb. 2024.

Simone, Sergio De. “Using Rust to Write Safe and Correct Linux Kernel Drivers.” InfoQ, InfoQ, 27 Apr. 2021, www.infoq.com/news/2021/04/rust-linux-kernel-development/.

Spafford, Eugene H. The Internet Worm Program: An Analysis - Purdue University, 29 Nov. 1988, spaf.cerias.purdue.edu/tech-reps/823.pdf.

Thompson, Clive. “How Rust Went from a Side Project to the World’s Most-Loved Programming Language.” MIT Technology Review, MIT Technology Review, 15 Feb. 2023, www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/.

Vaughan-Nichols, Steven. “Rust in Linux: Where We Are and Where We’re Going Next.” ZDNET, www.zdnet.com/article/rust-in-linux-where-we-are-and-where-were-going-next/. Accessed 2 Feb. 2024.