The Effect of Rust's Unsafe Mode on Developing a Security Mindset

A Research Paper submitted to the Department of Engineering and Society

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science, School of Engineering

Ratik Mathur

Spring 2024

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Advisor

Kent Wayland, Department of Engineering and Society

Introduction

As society and industry shift to a cyber-age, the internet has become a crucial component to global infrastructure. This includes mission-critical sectors such as healthcare or finance which provides a prominent attack vector for malicious actors to exploit any gap or oversight in software applications. Thus, defending against cyber-attacks has become an utmost priority for the modern-day software developer… or at least it should be.

Unfortunately, in practice we end up with insecure applications all the time which can often lead to ransomware situations such as the Ashley Madison data breach (Basu, 2015). This is often the result of a lack of security training. Looking at UVA, all security focused classes are electives; cybersecurity is not required for any computing program at UVA. Thus, developers are not even aware of what kinds of vulnerabilities to be wary of. Assuming a developer does have a security background, it is important to address perceived downsides from proactive security measures. One of the primary ones is that securing your app requires adding more lines of code which directly translates to a slower performance.

A solution to a lack of security awareness would be to build systems that have security baked in by default (such as in the operating system itself). However, such systems take the choice away from developers regarding speed and security and intentionally slow down the computer. Some types of applications can choose to be less secure for speed, especially in a testing environment or for personal projects. Some experienced developers may feel as if either they are forced into a security workflow that does not integrate well with their application or that their autonomy to optimize their own code is infringed on (Romeo, n.d.; Uluski et al., 2005).

Ultimately, it makes the computer take actions that are not exactly what the developer programmed.

Keeping that in mind, the question to ask is how can the trade-offs between performance and security in software development be navigated to address both technical and societal concerns, considering the impact on developers and the overall security landscape?

Rust is a relatively recent systems-level programming language that claims to offer a solution. By default, it enforces many restrictions on how code can access the computer's memory. However, it also supports writing code under what it calls "unsafe mode," where it relaxes many of the restrictions. This approach appears to combine both approaches in that by default it enforces security but allows a developer to relax restrictions if they know what they are doing, which overall results in Rust being a highly favored language in the community (*Stack Overflow Developer Survey 2016 Results*, n.d.). Thus, I conduct a literature review to analyze the effect of Rust's unsafe mode on developers' attitudes towards security and whether it has helped bring about a generally improved security mindset. In particular I wish to see if Rust serves as an effective solution to ensuring secure development without infringing on developers' autonomy.

Theoretically, a perfect developer would manage their memory properly and think of all edge cases. Obviously in practice this is not the case and thus many solutions were developed that arguably infringed on developer autonomy. But the necessity of secure systems in protecting the interest of society requires safeguards to be put in place which led to the creation of Rust as an alternative programming language. This approach to looking at Rust's development puts societal interest as the cause and Rust as a technology as a proposed solution to that problem. This warrants an analysis of the impacts of Rust through the lens of the SCOT (Social

Construction of Technology) framework. The overarching goal of this paper is to ask whether Rust serves as an effective tool for building secure apps with minimal infringement on Developer autonomy?

Technical Background

All (useful) languages have methods to deal with memory management, which means to either allocate (obtain) or deallocate (free up) some RAM to store program data. Many modern languages do this automatically by periodically checking all memory that has been requested and then deciding whether the program is done using it; this is called "garbage collection." This is necessary as dangling memory that is no longer in use can result in a whole class of memory corruption errors which provides numerous attack vectors for malicious actors and/or run the computer out of RAM. However, garbage collectors significantly slow down the execution of a program and the lack of one is one of many reasons why code written in C runs faster and it is thus used for operating systems and other performance-reliant apps. In C, the developer is responsible for manually freeing up memory after they are done using it. Unfortunately, proper memory management is very challenging even for skilled developers and thus the majority of system crashes happen due to memory corruption.

Do note that all programming languages are designed to be "Turing Complete" which is a set of conditions that theoretically prove that any program written in one Turing complete language can be replicated in *any* other Turing complete language. This means that regardless of unsafe mode or not, given infinite hardware, any Rust program can be written in C or any other language before that and vice versa. This concept further contributes to studying Rust from a social lens as it emphasizes that Rust is offering convenient abstractions to improve developer experience but

is not actually necessary from a purely technical perspective. This concept is expanded upon during the analysis of the collected literature.

Rust's central design concept is an ownership system in which every piece of memory has a well-defined owner, which would be a certain section of code, and once the owner is done executing, the memory is freed up. This eliminates the need for proactive garbage collection. C has no such concept and memory simply exists somewhere in RAM. By being very restrictive about memory accesses, Rust claims to eliminate entire classes of memory corruption errors. However, this restricts the types of programs that are allowed and Rust may reject programs (refuse to "compile" into binary) whose C equivalents are valid. This would make it unappealing for system developers to program memory managing applications such as an operating system. To get around this, Rust has another language built into it called "unsafe Rust." This relaxes some memory constraints and allows the programmer to use instructions that run the risk of introducing memory corruption. Additionally, it supports other unsafe variants, such as unchecked Rust, that skip various other security checks with the express intent of speeding up the application (*The Rust Programming Language*, 2023).

The principles behind this are to have potentially unsafe sections of code be thoroughly reviewed and be abstracted away to hide the unsafe code from other developers (such that they only use safe code). Rust supports a package manager, Cargo, where developers may upload their code in convenient packages, called Crates, for others to use. The intention is that the majority of unsafe code is restricted to crates with most application code never having to manually declare an unsafe section. Additionally, many types of programs would not be able to be written in Rust without unsafe mode (*The Rust Programming Language*, 2023).

The advantage of this form of system is that it enforces security by default, but allows a developer to relax such restrictions if they know what they are doing. This approach seems like a promising solution to ensuring secure applications are built while minimally impeding on developers' autonomy to make their decisions about speed and security.

Sociotechnical Situation

While the system being highly secure should be mandatory, both developers and project managers often push security to the side as something "nice to have" or as a "great add-on" (Lim et al., 2009). From the development perspective, implementing security requires significantly higher technical expertise than just simply creating an app. Often developers are not even aware of the current security landscape and thus do not even know what types of security they should be implementing (McGregor, 2019). Additionally, as developers often have security knowledge gaps (perhaps because they are either self-trained or their degree didn't teach them), app development teams often need security specialists if they want to secure their app which results in a higher development cost (Naiakshina et al., 2020). As far as managers are concerned, they want an app that runs smoothly with negligible latency. Similar to developers, they may also be blissfully unaware of the security needs of their app (Shreeve et al., 2023). An interesting concept presented by Lim, Joo S., et al. (2009) is that of OC: Organizational Culture. "OC refers to the systems of shared beliefs and values that develop within an organization and guide the behaviors of its members" and is typically formed by "the behaviors of dominant organizations members like founders and top management" (Lim et al., 2009, pg. 89; Schein, 1986). They discuss how their literature exploration suggests that developers' attitudes are largely developed by the culture of the organization they work for and how that culture often does not include a

security culture. This results in an environment that does not actively encourage security, which should be unacceptable for modern Internet-Scale Applications.

There seem to be three primary causes for insecure applications: awareness, functionality, and performance. There is an epidemic of a lack of awareness and understanding of the security landscape among developers which leads to insecure applications (Lopez et al., 2023). Even if they are aware of common vulnerabilities, developers may choose not to prioritize security as it does not directly impact the functionality of apps and would instead opt for adding more features. These two points come from perhaps some naïveté or general skill issues with developers. Improving the performance of the application is an area where a developer might explicitly choose to write a less secure application. This is because most defenses require adding additional lines of code which directly translates to an application that takes longer to run.

Prior Work

Rust makes bold claims about its ownership system in that it eliminates entire classes of errors (*The Rust Programming Language*, 2023). Many researchers have actually sought out to test these claims and how developers code in such a system. This has led to studies on Rust's type system, an empirical analysis of the Rust standard library, and even papers on operating systems arguing why Rust was chosen as the optimal choice in language (Astrauskas et al., 2020; Cui et al., 2024; A. N. Evans et al., 2020; Levy et al., 2017; Zhang et al., 2023).

These are all largely technical and empirical reports on the capabilities of Rust. This work will take a different angle in looking at what niche in society Rust exists to fill and how it has accomplished this task. I will be viewing their results under the lens of SCOT to help determine how developers are using unsafe Rust and how that aligns with the creators' intentions.

Theoretical Framework

I will be analyzing my chosen topic with the aid of Pinch and Bijker's STS theory known as SCOT: Social Construction of Technology. SCOT surmises that "culture and society shape technological development" as opposed to the traditional notion of "technology shaping society" (Pinch & Bjiker, 1987). Technological artifacts, according to SCOT, embody their cultural context, making this theory a useful lens for examining the role of Rust as a solution to the social issue of infringing on developer autonomy while also protecting the public's interests by developing a secure application.

Methodology

I collected various pieces from academia, largely empirical studies, that discuss the use of unsafe Rust by asking various research questions and will be reframing each of these sources as an answer to the research question of whether Rust fulfills a niche in the developer and larger public ecosystem to manage security and autonomy. I also present a counter argument to Rust mitigating security problems posed by a developer during a podcast and refute it with a Google blog on the use of Rust in Android.

Astrauskas et al. (2020) develop a testing framework that analyzes the content of various open-source Rust projects and a rather important metric they use is the percentage of unsafe code inside crates. Cui et al. (2024) propose a set of guidelines to describe an "unsafe Rust boundary" which describe what types of code should and should not be written in unsafe Rust. Evans et al. (2020) present similar research to Astrauskas et al. but focus on the limitations of Rust's memory checks due to much of the code implicitly using unsafe libraries at some point. Zhang et al.

(2023) focus on identifying common patterns of unnecessary uses of unsafe mode in Rust projects, that is code that has simple safe equivalents.

Results

A. N. Evans et al. (2020) surveyed a variety of crates on the central Rust repository, crates.io, to determine how much unsafe Rust was used both explicitly and implicitly (such as by importing a library that uses it). They sought to answer research questions that effectively measured how much unsafe Rust was used, how it was used, and why it was used. Importantly, they gather separate metrics for all crates overall as well as for the most popularly downloaded crates as they are likely the ones being used across multiple projects. This ended up making a rather large difference as overall 29% of crates had explicit uses of unsafe Rust, whereas 53.5% of the most popular crates had explicit unsafe Rust. Another important metric gathered, and arguably the biggest contribution of this paper, is that of those crates that contained no explicit unsafe code, 38% had dependencies that contained unsafe code thus making these crates also unsafe. They also looked at whether the Rust code was used to interact with data from C programs, as any such code would need to be marked unsafe, and found that only 22.5% (3.8% for most downloaded) of the unsafe sections were intended to interoperate with C. Thus, further suggesting that unsafe ought to be used for standard Rust libraries and may even imply that porting C code to Rust will not drastically affect the usage of unsafe as implementing such functionality would still require unsafe code. All of these metrics are from observing code, but they also directly interview Rust developers to see how they use the language. When asked about their reasons for using unsafe, 55% reported they used it for performance while 40% said it was because safe Rust was too restrictive; they also reported various other reasons such as language

interoperability and that they were generally more careful about reviewing unsafe code (A. N. Evans et al., 2020).

Astrauskas et al. (2020) look at whether unsafe code is used for libraries or for applications by also scanning packages on crates.io. This is done to answer a series of research questions that they hope will prove their "Rust Hypothesis" which states that unsafe code should be used *sparingly*, in a *straightforward* manner, and be well-hidden inside of crates. These research questions broadly encompass the following: how much unsafe code is used in crates, whether it is self-contained and, perhaps most importantly, what are the most prevalent use cases. This is similar to Evans et al. but here they are more focused on whether unsafe Rust is being used properly rather than observing whether unsafe Rust is used implicitly. While they present many valid use cases for using unsafe Rust, perhaps the most interesting is that they search for code that is marked as unsafe but actually contains only safe Rust code. "Since there is no technical reason why [these sections] need to be declared as unsafe, [they] expect that any such functions are declared unsafe either accidentally, or in order to document [the code]" (Astrauskas et al., 2020, pg. 136:11). Additional items they search for that are of interest are code that is indicated to be formatted to work with data that is produced by C code, often referred to as the C calling convention. The data overall shows that 76.4% of crates have no unsafe code within them, which is notably higher than Evans et al.'s study 10 months prior; these trends indicate that this percentage keeps rising as most valid uses of unsafe code are in already-developed standard libraries (Astrauskas et al., 2020).

Zhang et al. (2023) make important contributions in the form of a Visual Studio Code Extension (a plugin for a code editing environment) that suggests safe alternatives to unsafe Rust based on nine patterns they observed from crates.io. In particular they observed that there are

many recurring patterns of using unsafe code despite the standard library providing safe alternatives. Many of these alternatives are very similar in size and some are even more specific versions that can be guaranteed to be memory safe. They went further by inspecting unsafe code that has been reported to lead to bugs and found that their suggestions reduced such bugs by 28.6%, which is better than nothing and adds on memory safety guarantees. They also benchmark the performance overhead of swapping out unsafe code with safe code and find that generally there is a negligible change. But there were specific corner cases where there was either a slight improvement, likely due to better written code, and some where there was a significant slow down, likely due to enforcing further checks for memory safety (Zhang et al., 2023).

Cui et al. aim to describe a set of fine grained constraints, which they refer to as the unsafe Rust boundary, that describe under what conditions it is acceptable to use unsafe Rust and whether they can cover existing vulnerabilities caused by unsafe Rust. By auditing unsafe code in the standard library documentation, they constructed a list of 19 safety principles (SPs) and surveyed developers that have coded in Rust professionally about these principles. This classification helped them uncover that out of the 404 vulnerabilities they reviewed, 196 were due to unsafe Rust and 86.73% of those were caused by a misuse of the standard library. The responses from developers indicated that generally the SPs were precise (accurate) with a skew towards brief descriptions, the SPs were significant (important) with a skew towards memory safety, the SPs were usable (in the real world) despite having lower usability scores than significance scores, and the SPs came up frequently with a skew towards memory management. They also were able to map specific SPs to known Rust vulnerabilities (Cui et al., 2024).

Analysis

The perspectives taken generally by these papers are to analyze how developers are using Rust and whether that leads to memory issues. Each of these results can alternatively be used to assess Rust's efficacy as a solution to protect the interests of consumers, developers, and even business owners. C was developed in the 70s well before modern internet connectivity and mainly for designing software such as OS tools as current tools at the time were giving the creators of Unix a challenge, in fact many of C's design choices were to address frustrations developers had with then current languages such as B and BCPL (Ritchie, 1993). This lens portrays the development of C not as an artifact of technological advancement but as a solution to a social problem which is of frustration with then current tools, keeping in mind that C and its predecessors are Turing complete and thus all modern tools are (theoretically) achievable without C and its successors; in fact the creator states that "BCPL, B and C differ syntactically, but broadly they are similar [...] In spite of the differences, most of the statements and operators of BCPL map directly into corresponding B and C" (Ritchie, 1993, pg. 3). Taking such a lens is important as it reveals, at least partially if not more, why languages behave "under the hood" the way they do. C being designed primarily to cater to user experience when writing the source code meant that there was less of a focus on how it worked under the hood, whereas many modern languages emphasize how they operate behind the scenes as advertising. This left C as a prominent attack vector as we grew more interconnected. This displays why many modern languages take so many more actions implicitly, such as garbage collection, and even non-security issues such as portability between operating systems. And thus, we see Rust as a middle ground, attempting to obtain the best of both worlds in protecting the developer user experience as well as protecting society.

A. N. Evans (2020) et al. pointed out a rather important subset of Rust code, and that is the most popularly downloaded crates (a feature available on crates.io) as those libraries provide standard functionality that is expected to be used across most applications and are typically standard features in other languages. This includes functionality for interacting with the operating system or building complex data structures that rely on potentially unsafe memory operations to be implemented with reasonable efficiency. This subset is an important one to analyze as Evans et al. showed that most unsafe code is found here and this sample is used throughout much of the literature with many directly citing Evans et al.'s work.

Their work revealed that, despite the intention of unsafe Rust to be used for memory access not easily permitted by safe Rust, developers specified that they used it for execution speed ups more often than for using the additional memory access instructions it comes with. This shows two things: first that developers, when given the choice, are comfortable disabling certain security features (unsafe Rust does not disable *every* safety guarantee of safe Rust) to obtain a performance boost; second, in order to obtain a performance benefit, developers had to be aware of what safety checks were relaxed to reap their benefit as that specific, likely brief, code would have to be explicitly placed in an unsafe section. While the official Rust documentation doesn't necessarily encourage using unsafe for performance, it does acknowledge it as a benefit. Zhang et al. further highlight this point as some of their nine suggestions did incur a noticeable performance overhead while many remained negligible, which provides an avenue to make educated guesses about whether to emphasize security versus speed.

On the fourth episode of the Gamer Radio podcast, a show that invites professional developers and presents their attitudes towards development, the interviewee mentions that Rust's unsafe mode is simply offloading the issue of memory safety to a different abstraction

layer and does not really solve the problem (Dominick, 2023). A reddit post on this episode expands on this, with a user referring to it as an "unfamiliarity with Rust on a deep level" (echindod, 2023). Some of the results from Astrauskas et al., Cui et al., and Zhang et al. also confirm that there tend to be misuses of unsafe Rust which can lead to a lot of the problems with C. In fact, many of Zhang et al.'s safe alternatives to unsafe code consisted of replacing one line of code with a different line of code which shows a shortcoming of developers to read documentation closely. However, Cui et al.'s work shows that less than half of Rust vulnerabilities are due to an explicit use of unsafe code, which means that most vulnerabilities came from oversights in programming in safe Rust rather than by exposing unsafe memory accesses through unsafe Rust. Google further reports that they have been integrating Rust slowly into the Android ecosystem, not as a replacement for C/C++, but for developing new features because it is considered a memory safe language; at the time of writing their blog, "there [had] been zero memory safety vulnerabilities discovered in Android's Rust code" and the year it was written was "the first year where memory safety vulnerabilities [did] not represent a majority of Android's vulnerabilities" (Stoep, 2022). This is as expected since an alternative way of thinking about C is that the entirety of it is "unsafe" code whereas Rust only has certain sections that are as such.

Additionally, Rust is quite recent compared to C and has not been thoroughly integrated into the education sector unlike C which is prevalent throughout various departments outside of computing. Cui et al.'s work indicates that Rust is helping train developers in building a security mindset and they, as well as Zhang et al., developed tools (the SPs and VSCode extension) to help further educate Rust developers. UVA has also once taught their operating systems class in Rust (D. Evans, 2014). General Rust education should spark a greater security mindset as

knowledge of unsafe mode is necessary to build many foundational items (such as those found in the standard library) unlike with C where using it does not require learning security.

The term unsafe is generally a misnomer although the name does serve its purpose of alerting developers to look at the code carefully. Unsafe Rust just means that certain checks are being relaxed as safe Rust is over conservative in the programs it accepts, thus the intention is for the developer themselves to perform the safety checks and decide for themselves whether to accept the programs; unsafe Rust does not mean the code is unsafe, just that we have no deterministic algorithm to prove that it is safe.

I would advocate for baked in security because of a lack of developer awareness but present this work because I recognize the limitations it places on developers. To serve the needs of programmers, language developers should be leaving the choice between performance and security to the programmer and thus the solution for upholding the interests of society would be to leave this choice up to the programmers. However, the programmers in turn have a duty to society by building secure applications and this indicates that the solution that upholds society's interests is one with baked in security. This generates a self-conflicting sociotechnical system that Rust appears to alleviate. The ideal solution for this, which is obvious in hindsight, is to have security enabled by default and to let experienced developers who *know what they are doing* to disable these features so that they can tailor their code to the needs of their application. Without viewing baked-in security as a self-conflicting sociotechnical system, Rust's innovation seems diminished and for all we know it never may have been developed. Thus, I conclude that Rust has shown promise as an effective solution to provide developers with the autonomy to make decisions about performance and security tailored to their application needs while generally

inducing a security-focused mindset and directly reducing the amount of memory vulnerabilities plaguing industry.

Conclusion

Viewing Rust as a solution to a self-conflicting sociotechnical system simultaneously helps us view Rust's role in the larger developer ecosystem and provides a metric to measure its effectiveness at accomplishing said role.

Discussions of this nature are necessary as they underscore the lack of security integration in computer science education and the often-underestimated importance of security awareness for developers. They also highlight the challenges of balancing speed and security in programming languages, offering a glimpse into the decisions faced by developers when choosing the right tools and approaches. They also continue to add to the discussion of Rust as a memory safe language which in the literature appears as a very iterative effort.

I conclude that actual Rust code and developers' perspectives on Rust indicate an increasing security awareness, increasingly secure code, and a generally satisfied avenue to "take matters into your own hands" if the application needs demand it.

References

Astrauskas, V., Matheja, C., Poli, F., Müller, P., & Summers, A. J. (2020). How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 1–27. https://doi.org/10.1145/3428204

Basu, T. (2015, August 23). *Ashley Madison Slammed With $578 Million Lawsuit*. TIME. https://time.com/4007374/ashley-madison-578-million-lawsuit-canada/

Cui, M., Sun, S., Xu, H., & Zhou, Y. (2024). Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13. https://doi.org/10.1145/3597503.3639136

Dominick, M. (2023, January 22). *Alderon Games* (Vol. 4). https://gamerradio.fireside.fm/4

echindod. (2023, February 8). *Is rust `unsafe` just moving the problem?* [Reddit Post]. R/Rust. www.reddit.com/r/rust/comments/10x5s5k/is_rust_unsafe_just_moving_the_problem/

Evans, A. N., Campbell, B., & Soffa, M. L. (2020). Is rust used safely by software developers? *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 246–257. https://doi.org/10.1145/3377811.3380413

Evans, D. (2014). *cs4414: Operating Systems*. https://rust-class.org/index.html

Levy, A., Campbell, B., Ghena, B., Giffin, D. B., Pannuto, P., Dutta, P., & Levis, P. (2017). Multiprogramming a 64kB Computer Safely and Efficiently. *Proceedings of the 26th Symposium on Operating Systems Principles*, 234–251. https://doi.org/10.1145/3132747.3132786

Lim, J., Chang, S., Maynard, S., & Ahmad, A. (2009). Exploring the Relationship between Organizational Culture and Information Security Culture. *Australian Information*

Security Management Conference. https://doi.org/10.4225/75/57b4065130def

Lopez, T., Sharp, H., Bandara, A., Tun, T., Levine, M., & Nuseibeh, B. (2023). Security Responses in Software Development. *ACM Transactions on Software Engineering and Methodology*, *32*(3), 1–29. https://doi.org/10.1145/3563211

McGregor, L. (2019). Gamification and Collaboration to Evaluate and Improve the Security Mindset of Developers. *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, 342–343. https://doi.org/10.1145/3304221.3325593

Naiakshina, A., Danilova, A., Gerlitz, E., & Smith, M. (2020). On Conducting Security Developer Studies with CS Students: Examining a Password-Storage Study with CS Students, Freelancers, and Company Developers. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 1–13. https://doi.org/10.1145/3313831.3376791

Ritchie, D. M. (1993). The development of the C language. *ACM SIGPLAN Notices*, *28*(3), 201–208. https://doi.org/10.1145/155360.155580

Romeo, C. (n.d.). *Why developers dislike security—And what you can do about it*. TechBeacon. Retrieved April 29, 2024, from https://techbeacon.com/security/why-developers-dislike-security-what-you-can-do-about-it

Schein, E. H. (1986). Review of Organizational Culture and Leadership [Review of *Review of Organizational Culture and Leadership*, by W. G. Tierney]. *The Academy of Management Review*, *11*(3), 677–680. https://doi.org/10.2307/258322

Shreeve, B., Gralha, C., Rashid, A., Araújo, J., & Goulão, M. (2023). Making Sense of the

Unknown: How Managers Make Cyber Security Decisions. *ACM Transactions on Software Engineering and Methodology*, *32*(4), 83:1-83:33. https://doi.org/10.1145/3548682

Stack Overflow Developer Survey 2016 Results. (n.d.). Stack Overflow. Retrieved April 29, 2024, from https://survey.stackoverflow.co/2016

Stoep, J. V. (2022, December 1). Memory Safe Languages in Android 13. *Google Online Security Blog*. https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

The Rust Programming Language. (2023, February 9). https://doc.rust-lang.org/book/

Uluski, D., Moffie, M., & Kaeli, D. (2005). Characterizing antivirus workload execution. *ACM SIGARCH Computer Architecture News*, *33*(1), 90–98. https://doi.org/10.1145/1055626.1055639

Zhang, Y., Kundu, A., Portokalidis, G., & Xu, J. (2023). On the Dual Nature of Necessity in Use of Rust Unsafe Code. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2032–2037. https://doi.org/10.1145/3611643.3613878