CRusty Refactoring: An Evaluation of Refactoring Legacy C Systems to Memory Safe Rust.

Jack Sloane
Virginia Tech
sloanej@vt.edu

Matthew Jackson
Virginia Tech
mnj98@vt.edu

Ishaan Gulati
Virginia Tech
ishaangulati97@vt.edu

1 BACKGROUND

Rust is a young language with the motto "A language empowering everyone to build reliable and efficient software". This language is directed at speed, memory safety and parallelism. Rust was designed by Graydon Hoare, later funded by Mozilla Research, and is currently being developed by The Rust Foundation. It has grown in popularity ever since it first appeared in 2010 and subsequent significant releases in 2015 and 2018. The Stack Overflow surveys have shown Rust to be the "Most Loved Language" continuously over the past six years. It uses static typing, a multi-paradigm that supports imperative procedural, concurrent actor, object-oriented, pure functional styles, and smart pointers with reference to efficient memory management. Rust uses LLVM as its backend. Rust overcomes conventional programming language design conflict of "high-level ergonomics and low-level control that are often at odds with each other" by balancing developer experience and technical capacity [1].

For the past many years, C and C++ have been dominant languages for programming systems. C and C++ tend to compile directly into machine code, giving programmers control over what happens at the machine level. The programmer needs to expressly control how, where and when the memory is to be allocated. C boasts a minimal run time with minimal dependencies, making possible running of programs on smaller systems like embedded systems. Despite these advantages, C and C++ have some critical drawbacks in memory management causing severe zero-day vulnerabilities. Rust resolves problems C and C++ have struggled with, such as memory errors and building concurrent programs. Rust provides better memory management through the compiler, using a data ownership model to prevent concurrency data races and providing nearly zero-cost abstractions.

Rust has a feature called Zero Cost Abstraction, which was inspired by C++. Bjarne Stroustrup described the purpose of this feature, saying: "In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: what you do use, you couldn't hand code any better" [16]. In simpler terms, one does not pay for what one does not use, and high-level code compiles just as well as lower-level code. Hence, Zero Cost Abstraction implies that a higher level of abstraction would not incur additional costs during runtime. Rust uses the principle of Zero Cost Abstraction in traits, generics, iterators and most importantly, memory checks during compile time. Zero Cost Abstractions cause compilation time to increase as the compiler has to optimize the abstracted code.

For system programming, it is necessary to use low-level control provided by memory management. However, manual memory management comes with many issues in languages like C and C++. Despite having tools like Valgrid, memory management is tricky. Rust prevents memory management issues with the Ownership Model. Ownership Model moves the program's memory management to compile-time, ensuring that bugs due to poor memory management are spotted, which makes the garbage collection redundant. Furthermore, unsafe keywords can be used by programmers for better optimization similar to C. Data races happen when two threads simultaneously access the same data (memory), leading to unpredictable behaviour; Rust also prevents this undefined behaviour with Borrow Checker at compile time itself.

The ownership model makes Rust safer and better at concurrency. Usually, memory safety and concurrency bugs are caused by code assessing data that it should

not be accessing. Ownership provides a more stringent checking mechanism to access control. The stack manages primitive data types, i.e. things for which memory size is known, for example, Integer. Likewise, heap is used for items whose size might change (dynamic). The ownership rules dictate that:

(1) Each value in Rust has a variable called the owner.
(2) There can only be one owner at a time.
(3) When the owner goes out of the scope, the value will be dropped.

Whenever a value is created, it has an "owning scope". Passing and returning a value means transferring ownership to a new or different scope.

In Rust, every value has an "owning scope," and passing or returning a value means transferring ownership ("moving" it) to a new scope. Values that are still owned when a scope ends are automatically destroyed at that point. For example, a variable defined in a function gets deallocated automatically when the scope of that function ends. Returning a value from a function transfers its scope (ownership) to the calling function and passing the value as a parameter transfers its ownership to the function being called. If the value is not passed further to a function, then the value gets deallocated. One might question whether that value can be accessed in the parent scope; the answer is no, as once the ownership is given away, the value can no longer be used. But in the real world precisely this is not the case. One would like to continue using that value in the parent scope as long as the parent scope is still active. This is where "borrowing" comes in to lease the variable's to access functions that are being called. Rust will make sure that leases do not outlive the object that is borrowed. Leases are passed using a reference to the values, and references have a limited scope which is determined by the compiler. References are of two types [18]:

(1) Immutable References: Immutable references &T, which allow sharing but not mutation. There can be multiple &T references to the same value simultaneously, but the value cannot be mutated while those references are active [18].
(2) Mutable References: Mutable references &T, which allow mutation but not sharing. If there is an &T reference to a value, there can be no other active references at that time but the value can be mutated [18].

Rust checks for these rules during the compilation process and ensures that there is only one active borrow for the borrowed value. With the Ownership model, Rust ensures memory safety without using a garbage collector.

Due to the above exceptional features, Rust has caught the attention of many system programmers, including Linus Torvalds, who are strongly weighing in for using Rust for system programming. With this project, we want to explore various options for translating code written in C and C++ into Rust, including automated tools and manual rewriting and their efficacy.

2 PROBLEM FORMULATION

Now that we have covered some of the background information related to C, Rust, and memory safety, we can present the problem that we want to investigate for this project: How should developers go about refactoring legacy C code into memory safe Rust code? As we will elaborate further in a following section, we plan to look at the current research and methods related to automatic C to Rust translation tools. Then we hope to compare these methods with manually refactoring legacy C systems to Rust using metrics described in the Proposed Solution section.

In this section we will cover our motivation for researching this topic, our open questions about this topic, and the stakeholders invested in this technology. Each subsection should help clarify this problem and provide context for our proposed investigation.

2.1 Motivation

As we have discussed, the memory safety guaranteed by the Rust compiler is valuable in many ways. Due to Rust's ownership system, memory allocation is easier, less prone to bugs, and less prone to attacks. Ownership allows a safe dynamic memory system that does not require Garbage Collection since the compiler can detect when a pointer goes out of scope and automatically free its memory. There is a happy compromise where the memory safety doesn't come at the cost of Garbage Collection overhead.

Since Rust is clearly the best, safest, and most future proof language, it's obvious that systems developers have already completely switched over from C/C++ to Rust, and Rust is now the standard language for systems applications. Well, no, this is not the case. As of 2020

Rust had just made it into the top 20 programming languages according to the TIOBE index [17].

One issue for designing new systems in Rust is that the language is not considered to be easy to learn. A zdnet.com article titled "Programming languages: Rust enters top 20 popularity rankings for the first time" states that "Rust demands dedication to learn. Microsoft Azure developers initially were less productive in Rust than Go, but spent less time in the end debugging and manually checking for bugs that Go would have let pass through." Adoption of Rust has been slow since retraining a development team to use a new, and difficult, language may be too expensive in the long term.

However, refactoring existing memory-unsafe systems may be a more manageable task. For the purposes of this discussion we'll assume that a company is willing to expend some resources to refactor their outdated legacy system, but because developing with Rust may be expensive, they want to do this in the most efficient way.

Therefore what our group wants to investigate is if there are any automatic translation tools that are complete enough to use to update legacy systems to Rust. Additionally, we want to test how hard it is to manually refactor C code into memory-safe Rust code. If we can compare these methods we will hopefully be able to determine the best way to refactor legacy C code into modern Rust.

2.2 Open Questions

2.2.1 ***Manual Translation.*** Our group does not yet know how difficult it might be to manually translate C code into memory-safe Rust code. There could be many factors that contribute to how difficult or time consuming this process might be. It may be the case that translation is straightforward, yet we doubt this due to the different memory systems. Perhaps some functions need to be completely re-engineered to be compatible with Rust.

Additionally developers are expensive, and our group does not yet understand how much it might cost to either hire Rust developers or train current employees to use Rust. With this in mind the size of the legacy system to be refactored may matter a lot, and beyond some threshold it may not be feasible at all to manually refactor.

Our group does not yet know how difficult or expensive this process is, but we should be able to get more insight by researching how it is done in the industry and by translating some C code ourselves. The prevalence of C-to-Rust translation in the industry should give us insight into the tradeoffs that software companies consider, and our own experience should give some perspective on what it takes.

2.2.2 ***Automatic Translation.*** Based on our initial research for this deliverable we have identified that there are some tools for C-to-Rust translation, but we do not have a good understanding of all of the potential tools nor do we know how complete they might be. From what we have found, not all tools claim to cover all of the C language.

It is yet to be seen if any automatic tool produces quality Rust code. The concept of producing semantically identical Rust code seems trivial, but we are hoping for memory-safe Rust code which could prove to be a much more complex process. Verifying the correctness of automatically generated code may also be difficult.

We don't know how readable the automatically generated code is. With the TypeScript transpiler in mind, automatically generated code isn't necessarily very readable to a human. If the code is illegible it would be difficult to maintain, and the only benefit it might give is that it is memory safe.

Finally, we need to understand how easy an automatic translation tool is to use. The complexity of a tool's invocation or its running time could influence its practicality, or if there is a licensing fee to use it we may not be able to use a tool at all.

2.3 Stakeholders

This final subsection covers some of the stakeholders involved in this topic. Since software is replacing more and more parts of society, we could argue that everyone is a stakeholder in software issues. For the sake of simplicity we'll cover two more specific stakeholders: the software company updating the system and the developer as an individual.

2.3.1 ***The developer as a company.*** The company in charge of the project would have two primary concerns when it comes to refactoring a system written in C to one written in Rust. Firstly they want a product

that is of the highest quality possible, one that is readable, maintainable, and as performant as the original C implementation. However, quality must be balanced with cost, so there is an incentive to determine and implement the best technique for refactoring their legacy C system.

2.3.2 ***The developer as an induvidual***. The second stakeholder, the developer, would be more concerned with the difficulty of the project. The amount of effort required would be more important to a developer than cost or quality. A key quality of a good programmer is applied laziness, so ease of use is important to the developer. We will also argue that, in some ways, implementing quality code makes a developer's job easier because implementing the project correctly the first time reduces the cost of maintaining the system.

3 PROPOSED SOLUTION

For this project we will translate a set of C programs to Rust by hand and using multiple automated methods, compare the performance of the translations, and produce a written report of the results. All of the C programs we will translate embody classic programming problems. First are two sorting programs: Multikey quicksort and selection sort. Multikey quicksort is a three-way radix quicksort algorithm which sorts an array of N strings in lexicographic order. We chose to use sorting programs because they allow for a flexible input size. This will allow us to inflate the execution times, which may allow for more meaningful performance comparisons.

We chose these particular sorting programs for several reasons. First, these two algorithms are at opposite ends of the spectrum in terms of speed and time complexity. Additionally, the C implementation of multikey quicksort has many more lines of code to translate than selection sort. Lastly, the quicksort algorithm involves a heavy use of recursion, whereas selection sort is an iterative algorithm. We want to translate a diverse set of C programs in terms of execution speed, file-size, and logic because we hypothesize that the Rust programs outputted from C to Rust translation tools could perform differently depending on these characteristics.

In addition to the sorting programs, we will translate a client-server system from C to Rust, where the client-side and server-side implementations are separate C programs. We chose these programs in order to include a more practically relevant example in our analysis. Analyzing client-server programs adds a systems focus to our project that may widen the demographic of interest compared to a project solely focused on algorithms.

It is important to note that all the C programs we have chosen are atomic, using the standard C library "libc". No external libraries are required for their compilation and execution. This allows for a simpler and more direct translation from C to Rust when translating by hand. Additionally, the atomic nature of our C programs minimizes the degree to which the Rust code output from the translators rely on directly linking C libraries to the Rust output - a feature that Rust, in fact, supports.

As previously mentioned, this project involves translating C programs to Rust both by hand and using pre-existing translation tools. We will use two of the most predominant tools: C2Rust and Corrode (and perhaps others). Both translators produce unsafe Rust code that closely mirrors the input C code. In other words, the output of these translators is non-idiomatic Rust code that exactly captures the semantics of the C source file. In addition to Corrode, we will also analyze a related tool called Oxidize - a framework for idiomatic refactoring of Rust code. However, generating completely safe and idiomatic Rust code from C ultimately requires manual effort.

For a given C source program, we will compare the performance of the output from C2Rust, the output from Corrode, the output of Corrode refactored using Oxidize, our handwritten Rust Code, and the C source program itself as a control. We will evaluate performance in terms of memory usage and execution speed. We will additionally evaluate the client-server programs in terms of response time and data throughput (packets per second). We will perform these performance analyses predominantly using built-in Linux commands like "time" and "perf" which can profile processes for resource usage and execution speed respectively. We will use ApacheBench to measure the data throughput of the client-server programs.

In addition to quantitative analysis, we will record our overall experience using these translation tools, comparing them in terms of setup hassle, general ease of use, compatibility issues, etc. This project will culminate in the consolidation of our analysis into a written report. The main contents of this report will include

an introduction of our project and its motivation, descriptions and graphical visualizations of performance comparisons, our subjective experience using the tools analyzed, and a discussion of the implications of our findings.

4 BACKUP PLAN

There are a few ways that we may be unable to use or evaluate these tools. Firstly there is the possibility that a translation tool does not work as advertised, but this seems unlikely because the tools are mostly written by legitimate researchers. There is also the possibility that a tool runs in an environment that we do not have access to such as a rare Linux version or using software that requires licensing.

We may also encounter an issue if a tool does not fully support the C language. We have already identified that some tools have limitations, but we do not know to what extent. For example the C2Rust tool cannot understand `setjmp()` and `longjmp()` functions because their behavior is unclear.

One thing we must consider is the versions of C and Rust that these tools are compatible with. Should our original C implementations be written in C11 and the translation tool only supports C99, we could have an issue. Additionally an outdated translation tool could produce outdated Rust code.

C projects commonly import libraries through header files. We may be unable to access or translate imported libraries. This potential problem may only become relevant with larger scale projects which we are not planning on translating, but we will keep this in mind when thinking about the scalability of these tools.

In the case of missing underlying dependency, we might have to look for an alternative Rust package that supports a similar functionality as the missing dependency and translate the code manually. In the rare case, if we cannot find a similar Rust package, we would have to build up our implementation for that C package in Rust or link that C library using Rust crates likes Bindgen. In the second case, if the tool doesn't support the full C dialect, we might get partially or no output. To resolve this, we will have to translate the code into Rust manually.

We expect the translation tools to output working Rust code from simple C algorithms, but it is plausible that the translation tools could fail to translate the more complicated client-server programs directly to an equivalent and functional Rust client-server system. There may be incompatibilities across languages when it comes to support for networking functionalities like socket creation or connection. Hence, It may be necessary to significantly refactor the Rust code to the point where the underlying logic or algorithm is fundamentally changed. If this is the case, we would record these findings and express them in our project paper.

If we are not able to perform a comparative evaluation of client-server systems resulting from C to Rust translation, we will instead shift to a simpler C source program, and explain the reason why the client-server translation failed to work as expected in our final paper. In fact, if any translation items fail to function, we will do our best to explain the limitations of the translation tools that caused the problem.

5 LITERATURE REVIEW

5.1 The Rust Language

The paper discusses the design of Rust that makes it suitable for system programming. Rust supports concurrency and parallelism, which helps it in taking full advantage of modern hardware. Rust uses static typing and guarantees strong isolation, concurrency, and memory safety. One meaningful way it accomplishes this is by allowing fine-grained control over memory representations, with direct support for stack allocation and contiguous record storage. Rust uses its Ownership Model to ensure there are no data races and memory errors like double frees and dangling pointers. Each object in Rust has its owner. The ownership of an object can be handed over to the new owner, or the new owner can also borrow a reference to the object. Borrows can be either mutable or immutable. Mutable references can only have one owner at a particular time, i.e., only the active owner can modify them. On the other hand, immutable references can be freely copied, or new references can be created. This paper builds upon the need to translate unsafe memory code into Rust and discusses how the ownership model can help develop safe memory systems.[14]

5.2 How Do Programmers Use Unsafe Rust?

This paper does an empirical analysis of the open-source rust modules in the crates.io package registry to answer the questions: Are the developers following the principles laid out for using unsafe Rust. Usually, unsafe Rust is used for directly accessing hardware, manual memory manipulation and avoiding safety checks. The principles made by the Rust team were:

(1) Unsafe code should be used sparingly, in order to benefit from the guarantees inherently provided by safe Rust to the greatest extent possible.
(2) Unsafe code blocks should be straightforward and self-contained to minimize the amount of code that developers have to vouch for, e.g. through manual reviews.
(3) Unsafe code should be well-encapsulated behind safe abstractions, for example, by providing libraries that do not expose the usage of unsafe Rust (via public unsafe functions) to clients.

To answer whether these principles were followed, the authors analyzed nearly 29,000 packages manually and used an automated framework called Qrates. They found that the principle of using unsafe code sparingly was not followed. The authors had used MIR intermediate code representations to check the self-containment and straightforwardness of the code. The authors found that most of the unsafe code blocks had 21 lines and assumed that a small code length means lower code complexity. Therefore most of the unsafe code blocks are straightforward. They also found that most of the function calls inside a crate could be determined statically(nearly 82 percent), so the blocks are self-contained. The authors could not reach a definite conclusion for the third principle as most unsafe methods had public access but were mostly used for accessing hardware or another Rust library function.This work can further be extended in system applications like verified kernel extensions to, potentially, fully verified hypervisors, embedded OSs, etc.[3]

5.3 Learn Rust the Dangerous Way

Learn Rust the Dangerous Way (LRtDW) is a guide written by Cliff Biffle a former Google engineer that helps low level programmers understand Rusts's features. Biffle helps contextualize the differences between C and Rust for those who may not have a CS background. This is important because the differences between C and Rust are not purely syntactic, and understanding some of the theory behind these differences is crucial for writing idiomatic Rust code. Our group is interested in this guide because it covers the translation of the N-Body problem written in C into an idiomatic and memory safe Rust implementation. Biffle describes basic semantic translation, the difference between C pointers and Rust references, Rust optimizations, safe wrappers for unsafe functions, and finally how to remove the unsafe keyword entirely to produce idiomatic Rust that runs faster than the original C implementation. This guide should prove useful for when we attempt to implement our own Rust programs.[5]

5.4 Is Rust Used Safely by Software Developers?

This work performs a large-scale empirical study to explore how software developers are using unsafe Rust in real-world Rust libraries and applications. Their paper starts with a background on "unsafe rust", explaining that to allow access to a machine's hardware and to support low-level performance optimizations, a second language, unsafe Rust, is embedded in Rust. The results of their study indicate that software engineers use the "unsafe" keyword in about 30 percent of Rust libraries. In addition, more than half of the unsafe rust code cannot be statically checked by the Rust compiler because it is hidden somewhere in a library's call chain. The conclusion that this paper makes is that the claim of Rust as a memory-safe language may not be as realistic as is typically accepted. Our project is motivated in part by the allure of Rust as a memory-safe language - wherein developers are considering translating their existing systems from C to Rust, but they do not know the best way to go about it. According to this paper, common Rust libraries obscure a propagation of unsafeness that may circumvent the initial motivation for a transition from C to Rust. Hence, this paper relates to our project motivation by providing an honest perspective on the use of unsafe code in Rust.[7]

5.5 Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body

This work presents a comparative study between C and Rust in terms of performance and programming effort. Programming effort, in the context of this paper, refers to the challenge of writing high-performance computing programs, and the degree to which the code is maintainable and scalable. The paper describes how Rust emerged as a new programming language designed for concurrent and secure applications, adopting features of procedural, object oriented, and functional languages. The experimental work performed in this paper shows that it is indeed possible to establish that Rust is a language that reduces programming effort while maintaining acceptable performance levels. Both our project and this work involves performance comparisons between C and Rust. The findings of this paper validate the prerequisite motivation for our project which assumes that developers may desire to refactor their C systems to Rust. The difference between our project and this work is that we primarily aim to evaluate and compare the methods of translating C to Rust, instead of focusing on a performance comparison between C and Rust. Additionally, our project does not export the differences in programming effort between C and Rust.[6]

5.6 System Programming in Rust: Beyond Safety

This paper builds upon the safety applications of Rust and explores Rust's linear type system capabilities. Paper considers these capabilities in zero-copy software fault isolation, efficient static information flow analysis, and automatic checkpointing systems. The paper also discusses SingularityOs and how the Sing language manages memory. Rust's type system provides building blocks SFI (Software Fault Isolation) by ensuring a software component can also access memory objects granted by memory allocator or other software components, which Rust's Ownership model enforces. The authors implemented an SFI by adding support for the management plane to control domain lifecycle and communication by cleaning up and recovering failed domains, enforcing access control policies on cross-domain calls. Rust enables precise and effective Information Flow Control(IFC). IFC enables strong security by ensuring that sensitive information is not leaked to unauthorized channels. Most of the techniques for improving the performance and reliability of a system rely on the ability to manipulate the program's state in the memory. In Rust, each object has a unique owner; objects can be traversed easily. They are making it possible to easily maintain memory snapshots in applications such as transaction control, checkpointing, and replication. This work can further be extended in system applications like verified kernel extensions to, potentially, fully verified hypervisors, embedded OSs, etc.[4]

5.7 The Case for Writing a Kernel in Rust

The paper reports the experience of building a resource-efficient embedded in Rust with the minimal use of unsafe abstractions. The paper also argues how a linear type system in Rust will enable the next generation of operating systems. The article relies upon building the kernel in memory-safe language instead of relying on hardware protection. Building a kernel requires the use of unsafe code and a set of abstractions. The unsafe code includes code written by the Rust team in language libraries and kernel code written by OS developers. The collection of abstractions includes Cell, which Rust provides, and TakeCell, which the kernel provides. The paper also studies multiple kernel abstractions, including DMA, USB, Complex Data Structures, and Multicore. The article concludes that language-only techniques can mitigate the performance and granularity issues arising from hardware-enforced memory isolation. Only a small set of unsafe abstractions is necessary to form standard kernel building blocks. This paper helps in builds up our case stronger for translating C/C++ programs into Rust.[12]

5.8 Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study

This paper aims at supporting the greater adoption of Rust in particular and secure languages generally. To

better understand the benefits and challenges of adopting the Rust language, the authors conducted semi-structured interviews with professionals primarily at a senior level who have worked in Rust with their teams and a survey in the Rust developers community. From these surveys and interviews, the authors found that Rust has a near-vertical learning curve. Rust is hard to learn but has better compiler error messages as compared to other languages. Rust's official documentation provides a solid foundation with suitable examples. However, a few concepts like Ownership, Borrow Checking, and Lifetimes were hard to understand initially for developers. Nevertheless, once they understood the concepts better, they felt very comfortable and realized the mistakes they were previously making while writing unsafe code with other languages. The authors also discuss employers' concerns about using Rust for development, one of the stress points that most employers had was the small Rust talent pool. Key takeaways from the paper were how good documentation, community support, and feedback could help better Rust adoption and what changes can be made to flatten the learning curve.[8]

5.9 Rust as a language for high performance GC implementation

The work involves the implementation of an Immix garbage collector in Rust and C. The paper discusses how the choice of implementation language is a crucial consideration for when building a garbage collector. Typically, garage collectors are written in low-level languages like C or C++, but these languages offer little by way of safety and software engineering benefits. The paper describes how Rust's ownership model, lifetime specification, and reference borrowing deliver safety guarantees through a powerful static checker with little runtime overhead - and these features make Rust a compelling candidate for a garbage collector implementation language. The work presented in this paper shows that their Rust Immix implementation has similar performance to the C implementation, and that Rust's safety features do not create significant barriers for high performance. This work is similar to our project in that we both use low level C programs as a starting point, develop a Rust version of the program, and compare the performance - granted, the C programs we translate are much smaller scale then an entire garbage collector. Also, we are less interested in performance comparisons between Rust and C, and more interested in the performance comparisons between the Rust output of different translation tools.[13]

5.10 RustBelt: Securing the Foundations of the Rust Programming Language

Published in 2017, this paper attempts to prove the soundness of Rust's ownership system especially when it comes to core Rust libraries that contain unsafe code. To do this the authors reduced Rust to Rust which is "a formal version of the Rust type system . . . used to study Rust's ownership discipline in the presence of unsafe code."

While helpful, Rust's ownership system prevents certain functionality required for systems programming, so many Rust libraries contain unsafe code. This theoretically voids the memory safety guaranteed by Rust, but some people argue that well tested unsafe code can be safely encapsulated. This paper takes steps to prove this claim, and it does seem to hold for some libraries as long as identified preconditions hold. In addition to showing that various important Rust libraries are safely encapsulated, this paper also discovered a bug in Rust's standard library which demonstrates the usefulness of their model. [9]

5.11 Sandcrust: Automatic Sandboxing of Unsafe Components in Rust

This 2017 paper describes a Rust library created by the authors. Called "Sandcrust" this library allows developers to sandbox the execution of unsafe Rust code into a separate process which isolates the safe memory from unsafe operations. This is of interest to us as it is an alternative approach to maintaining memory safety while using unsafe code.

Traditionally a developer would void the memory safety of their Rust program if they used unsafe libraries, but Sandcrust allows developers to maintain memory safety in the main process. Sandcrust could be a workaround for developers who do not want to refactor their unsafe systems, but it is not a perfect solution. Firstly, Sandcrust does not prevent dynamic

semantic errors caused by unsafe memory because in the event of a bug it just returns the incorrect data back to the main process. The authors of the paper did not mention this, but the security vulnerabilities associated with unsafe memory are not addressed. This library seems to only protect against memory corruption. [11]

5.12 Citrus

Citrus is an "experimental C to Rust transpiler" with the goal to ease the C-to-Rust translation process. It was created in 2017 by GitLab user Kornel, and has not received a meaningful commit in 4 years.

Since it is considered an "experimental" tool, it has a small set of features and does not understand all C constructs. It cannot convert any arbitrary C code into Rust, nor can it even produce semantically equivalent Rust code. The point of the tool is to translate C syntax into Rust syntax, but it ignores C's semantics. Therefore the generated Rust code may not run nor compile correctly.

This leads to an output that must be manually refactored to be usable. We do not know the difficulty of refactoring after using Citrus, but there is an extensive guide on how to refactor both the initial C code and the outputted Rust code to streamline the process. This tool may be somewhat incomplete, but the guide itself might be of use to us when using other tools.

Our group has not yet attempted to run this tool.[10]

5.13 Oxidize: Framework for Idiomatic Refactoring of Rust Programming Language Code

This work involves the implementation of Rust transformation framework called Oxidize. The Oxidize framework transforms non-idiomatic Rust to idiomatic Rust code. This paper discusses how code transformation prompts the question of semantics preservation and validation of the generated code. For this reason, this research team implemented the Rascal Metaprogramming Language (MPL) - a syntax definition for the Rust systems programming language, together with the Rust transformation framework - Oxidize. Their research focuses on transformation cases like migration from the C style malloc memory management to Rust's ownership system, and idiomatic iterative statements transformations. The developers of Oxidize explain how the tool can be used in conjunction with C to Rust translation tools, specifically mentioning Corrode. In our project, we will attempt to utilize Oxidize to refactor Corrode-outputted Rust into idiomatic Rust. If successful we will evaluate and compare the performance of this Rust code separately from that of Corrode, C2Rust, and our hand-written Rust.[19]

5.14 C2Rust

C2Rust is a translation tool in development by companies Galois and Immunant. The tool is designed to be compatible with the C99 standard, and is meant to translate individual functions instead of whole projects to Rust.

This tool is designed to provide a semantic-preserving translation that should produce a compatible object file after compilation. However, it does not produce idiomatic nor memory safe Rust code, so we do not consider this tool to be a complete C to Rust translator. That is not to say that this tool has no use as it could be a stepping stone for a developer or a different tool to complete the translation.

The documentation for C2Rust is extremely good, and the developers, who give talks at Rust conferences and reply to issues and pull requests on GitHub, seem pretty accessible. This hopefully means that C2Rust is an easy to use tool that is well maintained.

Our group has only used the demo version of this tool.[2]

5.15 Corrode

Corrode performs automatic semantics-preserving translation from C to Rust. It is intended for partial automation of migrating legacy code that was implemented in C - it does not fully automate the process in the sense that the output is only as safe as the input, and the output largely lacks typical Rust idioms. Corrode should, however, produce code which is recognizably structured like the original C code, so that the output is as maintainable as the original. The compiled Rust output is ABI (application binary interface) compatible with the original C. Hence, if the Corrode-generated Rust is compiled to a .o file, it can be linked to exactly as if it were generated from the original C. Corrode is at the center of our project because it, together with C2Rust are the two C to Rust translation tools we will evaluate and compare.[15]

REFERENCES

[1] 2019. The Rust Programming Language. (2019). https://doc.rust-lang.org/book/ch00-00-introduction.html

[2] 2021. C2Rust. (2021). https://c2rust.com/

[3] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (Nov. 2020), 27 pages. https://doi.org/10.1145/3428204

[4] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 156–161. https://doi.org/10.1145/3102980.3103006

[5] Cliff Biffle. 2019. Learn Rust the Dangerous Way. (2019). http://cliffle.com/p/dangerust/

[6] Manuel Costanzo, Enzo Rucci, Marcelo Naiouf, and Armando De Giusti. 2021. Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body. (2021). arXiv:cs.PL/2107.11912

[7] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 246–257.

[8] Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. 2021. Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, 597–616. https://www.usenix.org/conference/soups2021/presentation/fulton

[9] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

[10] kornel. 2017. Citrus: C to Rust converter. (2017). https://users.rust-lang.org/t/citrus-c-to-rust-converter/12441

[11] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS'17)*. Association for Computing Machinery, New York, NY, USA, 51–57. https://doi.org/10.1145/3144555.3144562

[12] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys '17)*. Association for Computing Machinery, New York, NY, USA, Article 1, 7 pages. https://doi.org/10.1145/3124680.3124717

[13] Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2016. Rust as a Language for High Performance GC Implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/2926697.2926707

[14] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

[15] Jamey Sharp. 2017. Corrode. (2017). https://github.com/jameysharp/corrode

[16] Bjarne Stroustrup. 2012. Foundations of C++. (2012). https://www.stroustrup.com/ETAPS-corrected-draft.pdf

[17] Liam Tung. 2020. Programming languages: Rust enters top 20 popularity rankings for the first time. (2020). https://www.zdnet.com/article/programming-languages-rust-enters-top-20-popularity-rankings-for-the-fir

[18] Aaron Turon. 2015. Fearless Concurrency with Rust. (2015). https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

[19] Adrian Zborowski. 2017. Oxidize: Framework for Idiomatic Refactoring of Rust Programming Language Code. (2017). https://homepages.cwi.nl/~jurgenv/theses/AdrianZborowski.pdf