Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

Proposal for Thesis Research in Partial Fulfillment
of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Formal Verification of Safety Guarantees for Unsafe Code in a Rust-based Operating System

Elijah Rivera

December 2019

Supervisors:
Professor Howard Shrobe
Dr. Hamed Okhravi
Dr. Nathan Burow

Abstract

Most low-level software systems today are written in languages that have little to no enforcement of type safety and memory safety, resulting in a large amount of exploitable errors in systems.

Rust is a language with strict enforcement of type and memory safety, which allows it to guarantee that certain common memory errors (such as use-after-free or a null pointer dereference) cannot occur. However, this memory safety enforcement is too strict for some programming paradigms. For this reason, Rust has a backdoor in the form of the keyword `unsafe`, which allows the programmer to bypass the compiler's checks in certain places in the code.

Using this backdoor anywhere in a codebase compromises the Rust safety guarantees of the entire project, as the bypassed checks could propagate problematic changes even into safe code. This backdoor is also used in the implementation of some standard libraries of Rust, so code written on top of these libraries is also at risk.

The RustBelt project [6] was able to verify the safety guarantees of Rust code, excluding the use of `unsafe`. They further proved the semantic correctness of many of the standard libraries implemented with `unsafe`, and provided a way to extend this proof structure to other similar data structures.

Our proposed project will investigate the low-level programming paradigms found in the development of operating systems in Rust. We will attempt to

formalize a process for shrinking and/or removing instances of `unsafe` in the codebase. When we find instances of `unsafe` that are unavoidable, we intend to write abstractions around these uses that allow us to leverage the mechanisms of the RustBelt project to still prove correctness of the `unsafe` regions.

1 Introduction

Most modern-day low-level systems are written in C or C++. These languages have little to no enforcement of type and memory safety, nor protection against concurrency bugs.

Memory errors in programs are a major source of errors and exploitable vulnerabilities dating as far back as 1996 [10]. At BlueHat Israel 2019, Microsoft disclosed that in the past decade, memory errors have comprised $\sim$ 70% of discovered vulnerabilities in their products [9]. These errors include accessing memory after it has been deallocated/freed (also known as use-after-free), accessing memory before it has been initialized, accessing memory that is out-of-bounds (including the null address), and treating memory with a certain type as if it has a different type. Writers of C/C++ can easily accidentally write programs with any of these errors.

Concurrent programs also have additional opportunities for errors with data races, where a timing or scheduling difference could affect the final result of a program. These errors occur when two or more threads have

access to the same memory location and at least one of the threads is writing to that memory without using a proper locking discipline. C/C++ have no built-in protections against a programmer writing incorrect concurrent code. Incorrect concurrent code can be dangerous. A 2012 study at Columbia University showed that concurrency errors can also be exploited [13].

Reasoning about errors is difficult. Tools continue to be developed which help in detecting/mitigating these errors [4, 7, 11], but nonetheless memory errors and data races remain prevalent in operating systems. Memory safety bugs are frequently vulnerable to exploitation, leading in some cases even to attacks involving arbitrary code execution [3, 12].

A recent approach to solving this problem is found in the programming language Rust. The use of Rust is steadily increasing in the systems community due to safety properties inherent to the language design. Specifically, Rust claims that "you will never have to worry about type-safety or memory-safety. You will never endure a dangling pointer, a use-after-free, or any other kind of Undefined Behavior."[2] Rust's ability to make these claims relies on its novel approach to memory safety as type safety, using an ownership system. By encoding information about the kinds of reference to an object or the lifetime of the object into the type system, Rust is able to utilize existing compiler techniques to statically ensure that programs that compile meet the memory safety guarantees above.

Operating systems are pieces of critical infrastructure. When compromised, an adversary can gain access not only to the operating system but

potentially to all other software running on top. Writing an operating system in Rust provides memory safety guarantees to a security-critical part of computer infrastructure.

However, operating systems have operations that must necessarily break the Rust compiler restrictions. These operations include memory-mapped I/O operations, inline assembly code, direct pointer arithmetic and related instructions, and some other data structures with complex ownership requirements. These operations are prevented by different type-checker restrictions and therefore do not compile. For example, memory-mapped I/O operations often necessitate writing directly to a specific constant memory address. However, Rust's compiler prevents direct access to any specific memory address, to protect against mutating a piece of data that is potentially pointed to somewhere else in the program.

To get around these restrictions, Rust provides a backdoor in the form of the keyword `unsafe`. In Rust, `unsafe` signals to the compiler that the programmer is writing code that he/she knows will not pass the Rust type-checker. The burden of verifying that the code adheres to memory-safety and type-safety falls back onto the programmer. This means that code that includes a dependence on `unsafe` ultimately has the same level of guarantee as pre-Rust solutions: "hopefully the programmer is correct."

Unfortunately, this backdoor doesn't just undermine the guarantees of code that contains `unsafe` section inserted by the programmer. Since `unsafe` forces the compiler to ignore certain checks, any values modified within or

returned by `unsafe` code could break all type and memory safety guarantees, even once passed to safe code. Once `unsafe` is used, everything it transitively interacts with is also necessarily `unsafe`. What's worse is that many of the Rust standard library data structures also contain some amount of `unsafe` code. Any code that relies on these data structures also now has a similar lack of the earlier formal guarantees. Now even though we started with a language that is statically safe, even using the standard libraries can potentially violate this safety.

The RustBelt project [6] was able to formalize a large subset of Rust and proves that its semantics are correct and do uphold the claimed guarantees. The project began with Safe Rust, which is just Rust without any uses of the `unsafe` keyword.

RustBelt formalized a subset of Rust's mid-level intermediate representation (MIR) in Coq, and then produced machine checked proofs that verify correctness. It then goes further, and provides a method for proving correctness for new libraries which depend on `unsafe` code. It uses this method to verify correctness for a large portion of the standard libraries. This enables us to once again confidently assert the type-safety and memory-safety guarantees of Rust's static type-checker.

But this confidence is limited. These proofs still only apply to programs written in Safe Rust whose dependencies are limited to the standard libraries checked by RustBelt. In the world of operating systems, this is simply not enough.

In this project, our goal is to extend the confident guarantees of Safe Rust to low-level systems programming paradigms. We will take an operating system written in Rust as an example, categorize patterns of necessary usage for `unsafe` code, and attempt to statically verify that this usage still adheres to the safety guarantees of Rust mentioned previously.

Depending on the patterns encountered, the needs of this guarantee may differ. For many of the internal data structures we expect the process to look similar to the extensible approach taken by RustBelt, while we expect new approaches to be necessary for other patterns of `unsafe` usage.

One of the first ways to increase confidence in the code is to decrease the number of places where `unsafe` is used. Thus, before attempting to make claims about necessary `unsafe` code, we will attempt to isolate and reduce the usages of `unsafe` where possible. Reducing the number of locations of `unsafe` restricts the number of places where the burden is on the programmer to ensure safety, and allows the compiler to take the burden of safety in all other sections.

2 Background

Before describing our proposed contribution and its effectiveness, we first will discuss pertinent background information on the memory safety of Rust, the successes and limitations of the RustBelt project, and the operating system Tock [8] which we will be using for our analysis.

2.1 Safety of Rust

Rust's approach to memory safety involves the use of an extensive ownership system integrated into its type system. This ownership system is able to provide both memory and concurrency safety.

In this system, variable bindings "have ownership" of the data they're bound to [1]. For this reason, Rust also prevents more than one variable from being bound to a piece of data. Reassignment is processed with move or copy semantics, instead of aliasing as in many other programming languages.

Rust also has the notion of "borrowing", which creates a reference to something without taking ownership of it. Even though only one variable can have ownership of a piece of data, many borrowed references can exist to the same piece of data. There are both mutable and immutable references, and the type system enforces that you do not have more than one reference to a piece of data when one of those references is mutable. This enables the type system to statically check for and prevent data races at compile time.

A reference also has an associated lifetime marked in the type system. The lifetime of the reference is never allowed to exceed the lifetime of the variable binding itself. This prevents use-after-free errors.

These ownership rules are formalized in the RustBelt system and proven correct for its formalization of Safe Rust. However, they are not guaranteed to hold in places where `unsafe` is used. RustBelt attempts to reason about these sections of code with the ideas of syntactic and semantic correctness.

2.2 Syntactic and Semantic correctness

Most systems programmers are not trying to write incorrect code. The programmer has some sort of internal intuitive argument for why their code is correct and type-safe, even though the compiler may disagree.

One canonical example of this is interior mutability, or the idea of mutation through a shared reference. This is explicitly forbidden by the ownership rules above, which state that no more than one reference to a piece of data can exist if that one reference is mutable. However, this functionality is still available in Rust, through a standard library construct called Cell.

By necessity, Cell is implemented in Rust using the `unsafe` keyword. However, the RustBelt project was still able to prove that its API adhered to the restrictions of Safe Rust. That is, through careful restrictions of `unsafe` usage, the code still behaves like safe code (semantic correctness), even though it doesn't adhere to the strict rules of the type-checker (syntactic correctness).

2.3 RustBelt for Unsafe Rust

RustBelt provides a way to specify what must be proved in order to confirm that a library API actually does constitute a sound extension of Rust's type system. Given a description of the API, it will generate obligations that the user must prove hold before they can say that the described API is safe to use.

There is also included machinery for determining that implementations of the API actually adhere to the typing specifications provided, and examples of this in the proofs of the standard libraries.

2.4 Tock

Tock is an embedded operating system written entirely in Rust to take advantage of the additional safety properties Rust provides over C/C++, which most other operating systems are written in.

Within this operating system, there are a number of data structures with safe APIs used by the rest of the codebase. These data structures should be able to be proven semantically correct with minor addition to the RustBelt framework.

However, this only covers one type of `unsafe` usage in our operating system code, and is not enough for many other paradigms. Both inline assembly code and memory-mapped I/O operations directly access and modify memory addresses, which Rust's ownership type system cannot reason about. As RustBelt only serves to prove the guarantees of Rust's type system, it will be unable to make any safety guarantees about these types of `unsafe` code. We want to be able to guarantee the safety of the entire operating system, including these sections of code, and so additional methods, formalizations and proofs will be necessary.

3 Proposed work

The first part of the project will be to locate and isolate instances of `unsafe` in Tock, with the goal of either eliminating the need for `unsafe` entirely or at least shrinking it to the smallest possible size.

Writers of operating systems are aware of the restrictions of Rust and the necessity of `unsafe` in operations. However, this sometimes results in unnecessary use of `unsafe` when the safe way to write the low-level code is not obvious. As a preliminary exploration, we attempted to rewrite one of the core data structures in Tock with only safe code, and were successful. This data structure is called TakeCell, and essentially acts as a wrapper around a Cell with a potentially missing mutable value inside. The TakeCell API also provides convenient interfaces for the data which are not directly offered in the Cell API. The `unsafe` code for many methods API can be reduced to simple single-line programs, while the safe version is $\sim$ 5 lines and is non-intuitive.

We suspect that there are other instances where similar tradeoffs exist. We expect to be able to further reduce or eliminate `unsafe` code in other similar data structures in the near future, and to be able to generate documentation to help developers understand how to accomplish similar tasks without needing to default to `unsafe`.

As we encounter irreducible blocks of `unsafe` code, we will document and categorize them. We hope to see duplicated patterns that allow us to factor

out many `unsafe` uses into shared libraries with safe APIs, which would allow us to leverage the RustBelt machinery to prove these pieces correct.

We fully expect to encounter paradigms of `unsafe` use that do not fall into the previous categories of reduction or abstraction. For instance, we do not expect it to be possible to reduce inline assembly, nor to abstract it away. Often times these sections of assembly code are needed to perform specific hardware-dependent operations that are beyond the abstract scope of Rust's safety model. These sections may not adhere to Rust's syntax OR semantics, as assembly is a different programming language in itself. In these cases, we hope to at least be able to isolate the `unsafe` code, and write proofs (maybe handwritten but preferably machine-checked) about the correctness of such sections.

4 Timeline

- January 2020 - rewrite of existing Tock data structures to eliminate/minimize use of `unsafe`
- February-March 2020 - formalize remaining Tock data structures in the RustBelt system and prove them semantically correct
- April-May 2020 - isolate other `unsafe` codeblocks and create common abstractions/data structures to represent operations performed
- June-August 2020 - off for the summer

- September 2020 - formalize new abstract data structures in RustBelt if possible, in Coq with some other abstraction if not
- October-December 2020 - prove semantic correctness of new abstract data structures
- Spring 2020 - evaluation, testing, and write-up

5 Alternative approaches

This is not the only way we could approach adding safety to an operating system. Here we will discuss other approaches that have been taken and the trade-offs they have.

5.1 Improved static checking

Instead of only addressing the current uses of `unsafe` in the codebase, we could instead attempt to improve the overall static type-checking capabilities of Rust, perhaps incorporating some of the reasoning about syntactic and semantic correctness.

Unfortunately, current systems have a difficult time with automated reasoning about programs without a significant additional burden on the programmer. Rust already has a more steep learning curve with the inclusion of lifetimes in the type system, and to include even more may present an undue burden on the programmer.

5.2 Software testing

We could instead use standard bug-finding techniques to find errors in the system, including writing large test suites with many edge cases and using fuzzers.

We actually recommend this approach be pursued in combination with the proof techniques in our proposed work. While having proofs of correctness offers guarantees of safety that one cannot acquire otherwise, these proofs will always rely on assumptions of the system. Formally verifying a system is not enough to guarantee that it is error-free[5]. Bug-finding tools can serve to spot-check and make sure the system assumptions hold.

5.3 Error detection/mitigation

Instead of the above compile-time options, checks can be included at runtime to detect when errors have occurred and attempt to mitigate the damage done.

This again has no guarantees of safety. However, in places where proof boundaries exist, having runtime instrumentation to check that our assumptions haven't been violated can serve to reinforce our confidence in the safety of the system.

One specific place where this can be seen is the hardware boundary.

6 Summary

Memory errors are a frequent and dangerous occurence in operating systems. The programming language Rust makes the strong guarantee of memory safety under normal conditions. These guarantees are broken when the `unsafe` escape hatch is invoked. This escape hatch is often necessary for different operating system programming paradigms. Our proposal will verify that these instances of `unsafe` in a specific operating system (Tock) do not break the memory safety guarantees of the languages, and thus we will prove that the operating system is itself memory-safe.

7 References

References

[1] The rust programming language.

[2] The rustonomicon.

[3] Multiple vulnerabilities in google android os could allow for arbitrary code execution. https://www.cisecurity.org/advisory/multiple-vulnerabilities-in-google-android-os-could-allow-for-arbitrary-code-execution_2019-088, Sept. 2019.

[4] Criswell, J., Geoffray, N., and Adve, V. S. Memory safety for low-level software/hardware interactions. In *USENIX Security Symposium* (2009), pp. 83–100.

[5] Fonseca, P., Zhang, K., Wang, X., and Krishnamurthy, A. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 328–343.

[6] Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages 2*, POPL (2017), 66.

[7] Kuvaiskii, D., Oleksenko, O., Arnautov, S., Trach, B., Bhatotia, P., Felber, P., and Fetzer, C. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 205–221.

[8] Levy, A., Campbell, B., Ghena, B., Giffin, D. B., Leonard, S., Pannuto, P., Dutta, P., and Levis, P. The tock embedded operating system. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems* (2017), ACM, p. 45.

[9] Miller, M. Trends, challenges, and shifts in the software vulnerability mitigation landscape, Feb. 2019.

[10] One, A. Smashing the stack for fun and profit. *Phrack magazine 7*, 49 (1996), 14–16.

[11] Seyster, J., Radhakrishnan, P., Katoch, S., Duggal, A., Stoller, S. D., and Zadok, E. Redflag: A framework for analysis of kernel-level concurrency. In *International Conference on Algorithms and Architectures for Parallel Processing* (2011), Springer, pp. 66–79.

[12] Shacham, H., et al. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM conference on Computer and communications security* (2007), New York,, pp. 552–561.

[13] Yang, J., Cui, A., Stolfo, S., and Sethumadhavan, S. Concurrency attacks. In *Presented as part of the 4th {USENIX} Workshop on Hot Topics in Parallelism* (2012).