Verifying Vulnerability Fixes in a Rust Verifier

Bachelor Thesis Project Description

Olivia Furrer

Supervisors: Vytautas Astrauskas, Jonas Fiala, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zurich

March 2023

1 Introduction

Rust is a systems programming language that guarantees memory safety at compile time by default. This property is based on three core concepts: *ownership* of memory, *borrowing* and *aliasing xor mutability*.

Each value in Rust has an *owner variable*. As soon as the owner variable goes out of scope, the memory used by its value is reclaimed immediately. During the lifetime of the owner variable, Rust allows the *borrowing* of a value, i.e. the creation of a reference to it. These references are not allowed to *outlive* the owner variable, which prevents traditional safety issues like use-after-free or dangling pointers. Rust's *aliasing xor mutability* property ensures that the two types of borrowing, namely *shared borrowing* for read access and *exclusive mutable borrowing* for write access, are never present at the same time. This makes concurrent reads and writes impossible in Rust and thus prevents conventional race conditions and memory safety bugs like accessing invalid references.

While these safety rules guarantee that no undefined behaviour can ever be caused by *safe* Rust code, they are too restrictive to model some of the low-level hardware behaviours that are required for system software. For this reason Rust introduces the `unsafe` keyword. It is used to temporarily delegate the responsibility of ensuring memory safety in the code to the programmer. If an API contains unsafe code, its author could choose to directly expose this internal unsafe code to the users of the API. However, it is considered more idiomatic to encapsulate internal unsafe code with a *safe* API. When a programmer declares an API as safe, they assure that the API conforms with Rust's safety rules. This means that the programmer is responsible for ensuring that (1) no matter what input the API is given by a client, no memory safety bug can be triggered, and (2) any internal unsafety hidden in the safe API is properly guarded. Since the Rust compiler cannot conclude the safety of unsafe sections by itself, it relies on the programmer and assumes that the code is sound and bug-free in order to include it in the program's safety guarantee. However, if this assumption is wrong and there is in fact a bug in unsafe code, this can result in the safety guarantee of the entire Rust program being compromised. Unfortunately, reasoning about the correctness of unsafe code is very hard for programmers, as one often has to consider actions taken by the Rust compiler that are not visible to the programmer. For example, when reasoning about panic safety, a programmer needs to manually check the consistency of stack variables for every (invisible) unwinding path which is inserted by the compiler automatically [1].

For this reason, tools are built to automatically analyze code. One strategy is to search for bugs and remove them, another strategy is to prove the correctness of the code given.

One recently developed program which searches for bugs in Rust code is RUDRA [1], written to specifically target `unsafe` sections. RUDRA is programmed to recognise certain patterns in the code that have been found to often appear in connection with memory safety bugs. It scans Rust packages and marks sections where these patterns appear, indicating a *potential* memory safety bug to the programmer.

RUDRA's developers scanned and analysed the entire Rust package registry using RUDRA. They discovered 264 previously unknown memory safety bugs. The 112 RustSec advisories they filed correspond to 51.6% of memory safety bugs reported to RustSec since 2016 (until the time of the publishing of their paper in 2021). Among them were bugs in some of the most often used Rust packages (e.g. in `std`, the Rust standard library, or in `rustc`, the Rust compiler). Some of them had gone undiscovered for years despite these important packages being written and thoroughly reviewed by Rust experts, and despite the Rust community's efforts to manually audit unsafe code in Rust [1].

There is, however, one major drawback of programs like RUDRA that search for bugs: false negatives. Even if we do not find a single bug in a program, that does not mean that there is none. There might still be bugs which go unnoticed, undermining the security of the entire program, or even a whole system.

The problem of false negatives is remedied by a different approach where, instead of searching for bugs, one tries to prove the correctness of a program (and thus the absence of bugs). An automated verifier takes this approach and is designed to prove that a given program conforms with its specifications. An example is Prusti [2], an automated program verifier which is currently being developed at ETH Zürich. It is built upon the Viper [3] verification infrastructure (also developed at ETH) and is targeted at the Rust programming language. Prusti allows the user to provide specifications through various features and checks whether they can be proved for the given piece of code [4].

When Prusti's development began, the focus was on *safe* Rust code. Increasingly now, the goal is to expand Prusti to also target certain patterns of *unsafe* Rust code.

The aim of this project is to evaluate how well Prusti is suited for verifying the absence of vulnerabilities caused by memory safety bugs.

2 Approach

Some of the bugs which were found by RUDRA have been fixed since they were discovered. We want to see whether these fixed versions of the code can be verified by Prusti in its current state of development. If there are not enough fixed RUDRA bugs suitable for our evaluation, we will expand our set of examples by fixing some of the unpatched bugs ourselves in order to include them, or by including bug fixes from other sources.

As a preparation, we get an overview of which Rust features are present in the relevant sections of the code. If any of them are features which we know are not supported by Prusti, we try to rewrite the code in a way which avoids these unsupported features but does not change anything relevant to the bug and its fix. For some code examples this might not be feasible, meaning we will not be able to verify these examples. For the examples where we expect verification with Prusti to work, we write specifications in Prusti and let Prusti run on them. If Prusti verifies our example successfully, we need to make sure that, using the same specifications, Prusti is *not* able to verify the buggy version of the code. Being able to verify the buggy version indicates that either we made a mistake in our specifications, or there is a bug in Prusti. If our specifications are buggy, we need to correct them and repeat the verification of first the fixed and then the buggy version. In cases where, contrary to our expectations, Prusti fails to verify the correct version, we find out whether this is caused by Prusti missing some feature which would allow it to verify our example, or by some other bug in the code, or whether there is some other reason for the failed verification.

2.1 Examples

2.1.1 Example 1: The retain function of std::string::String

Figure 1 shows the `String::retain` function from the Rust standard library, where a panic safety bug has been found, including the fix of this bug. The `retain` function is meant to go through a string and retain only the characters specified by the predicate `f`. If there are characters to be left out, followed by ones to be retained, the copy function on lines 29 to 33 copies the latter ones to their new spot further at the front. String has a type invariant stating that it must always be a valid UTF-8 encoding. Since UTF-8 is a variable-length encoding, this invariant gets temporarily broken during the execution of the while loop, as the characters being copied do not always have the same length in bytes as the one to be overwritten. This constitutes a problem as the predicate `f` (line 25) could panic and unexpectedly exit the code, exposing the non-UTF-8 string outside of the function.

In order to solve this, the length of the string was forced to zero before entering the while loop (line 19). Now, if `f` causes the function to be exited unexpectedly, the string will have length 0

```
        use std::ptr;

        pub struct String {
            vec: Vec<u8>,
        }

        impl String {
            /* ... */

            pub fn retain<F>(&mut self, mut f: F)
            where
                F: FnMut(char) -> bool,
            {
                let len = self.len();
                let mut del_bytes = 0;
                let mut idx = 0;

/* + */         unsafe {
/* + */             self.vec.set_len(0);
/* + */         }

                while idx < len {
                    let ch = unsafe { self.get_unchecked(idx..len).chars().next().unwrap() };
                    let ch_len = ch.len_utf8();
                    if !f(ch) {
                        del_bytes += ch_len;
                    } else if del_bytes > 0 {
                        unsafe {
                            ptr::copy(
                                self.vec.as_ptr().add(idx),
                                self.vec.as_mut_ptr().add(idx - del_bytes),
                                ch_len,
                            );
                        }
                    }
                    // Point idx to the next char
                    idx += ch_len;
                }

/* - */      /* if del_bytes > 0 {                                                               */
                    unsafe {
                        self.vec.set_len(len - del_bytes);
                    }
/* - */      /* }                                                                                */
            }

            /* ... */
        }
```

Figure 1: Preventing `String::retain` from creating non-UTF-8 strings when abusing panic to fix a panic safety bug in `std::string::String` [5] [6].
(CVE-2020-36317 [7])

and the UTF-8 invariant will hold trivially. On the other hand, if the while-loop is completed successfully, the length is set to the real length of the result (line 42).

Now that we have analysed and understood both bug and fix we make a list of Rust features present in this code:

- structs: `String`
- closures: predicate `f` handed to the `retain` function
- traits: `FnMut`
- slices: returned by `get_unchecked(idx..len)`
- iterators: returned by `chars()`
- enums: `Option` returned by `next()`
- raw pointers: `ptr::copy(...)`
- ...

Many of these features are not yet supported by Prusti, e.g. closures, slices and iterators. As we simplify the code in preparation for verification, we try to rewrite it in a way that circumvents these unsupported features. In figure 2, for example, we replaced the closure (which the `retain` function was previously given as an argument) by a new function `f(ch:char) -> bool` outside of `retain`. We also created a new, unimplemented function `get_next_char()` for simulating getting a next character without having to use slices and iterators, which Prusti does not support. As a third modification we now hand the function a vector as an argument directly instead of handing it `self`. This allows us to simplify the rest of the code by replacing all uses of `self.vec` by just `vec`.

In this example, even more modifications would be needed before it can be handled by Prusti; in its current state there are still some issues we need to handle during the course of our project to make this example verifiable.

Once this is done and our code is ready, we will write Prusti specifications for its verification. For now, we simply give some rough ideas of what we might do in this step: We could annotate our new `get_next_char()`function with `#[trusted]` since it does not matter to us what this function does exactly, we are only interested in the fact that a char is returned. Another possibility is to add external specifications like the ones on `set_len` shown in figure 3.

After having written our specifications, we run Prusti and see whether our verification is successful. Depending on the result, we decide what steps are to be taken, according to our approach as described above.

2.1.2 Example 2: Implementation of Random for arrays in the autorand package

Figure 4 shows the fix of another bug found by RUDRA, this time in Rust's `autorand` package[1] [8] [9]. Here, the following features are used:

- traits (e.g. `autorand::Random`)
- union `std::mem::MaybeUninit`
  and associated functions `uninit()` and `assume_init()`, the latter of which is unsafe
- unsafe function `std::mem::transmute_copy`, which reinterprets memory as a different type

Verification of the fixed version is unfortunately not possible here, since both `std::mem ::MaybeUninit` and `std ::mem::transmute_copy` rely on reinterpreting memory. This is a feature which is currently not supported by Prusti, and when we try to verify the fixed code from figure 4, we get the output shown in figure 5. The code cannot be rewritten to avoid these features since, in this case, they constitute the bug fix itself. Omitting or replacing them would defeat the point of verifying this fix. Therefore, verification is not feasible in this case.

[1]Where we instantiated the macro by setting the macro variable to 10 for simplicity.

```rust
use std::ptr;

fn f(ch: char) -> bool {
    unimplemented!();
}

unsafe fn get_next_character() -> char {
    unimplemented!();
}

pub fn retain(vec: &mut Vec<u8>) {
    /* ... */

    let len = vec.len();
    let mut del_bytes = 0;
    let mut idx = 0;

    unsafe {
        vec.set_len(0);
    }

    while idx < len {
        let ch = unsafe { get_next_character() };
        let ch_len = ch.len_utf8();

        if !f(ch) {
            del_bytes += ch_len;
        } else if del_bytes > 0 {
            unsafe {
                ptr::copy(
                    vec.as_ptr().add(idx),
                    vec.as_mut_ptr().add(idx - del_bytes),
                    ch_len,
                );
            }
        }
        // Point idx to the next char
        idx += ch_len;
    }

    unsafe {
        vec.set_len(len - del_bytes);
    }

    /* ... */
}
```

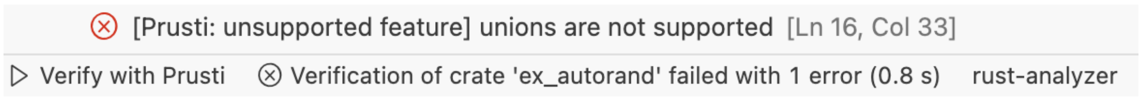
Figure 2: Rewriting and simplifying the code from figure 1

```rust
#[extern_spec]
impl<T,A: std::alloc::Allocator> std::vec::Vec<T,A> {

    #[pure]
    pub fn len(&self) -> usize;

    #[ensures(self.len() == new_len)]
    pub unsafe fn set_len(&mut self, new_len: usize);
}
```

Figure 3: External specifications for `std::vec::Vec::set_len`, intended for verification of the `String::retain` function

```
        pub trait Random: Sized {
            fn random() -> Self;
        }

        impl<T: Random> Random for [T; 10] {
/* + */     fn random() -> Self {
                use std::mem::{MaybeUninit, transmute_copy, size_of};
                unsafe {

/* - */          /* let mut array: [T; 10] = std::mem::uninitialized();                        */
/* - */          /* for i in 0..10 {                                                           */
/* - */          /*      std::ptr::write(&mut array[i], T::random());                          */

/* + */             let mut array: [MaybeUninit<T>; 10] = MaybeUninit::uninit().assume_init();
/* + */             for elem in &mut array[..] {
/* + */                 *elem = MaybeUninit::new(T::random());

                    }

/* - */          /* array                                                                      */

/* + */             // See https://github.com/rust-lang/rust/issues/47966
/* + */             debug_assert!(size_of::<[MaybeUninit<T>; 10]>() == size_of::<[T; 10]>());
/* + */             transmute_copy::<_, [T; 10]>(&array)
                }
            }
        }
        // from autorand-rs/src/lib.rs
```

Figure 4: Fix of a bug found by RUDRA in the `autorand` package [8] [9] (RUSTSEC-2020-0103 [10])



```
ⓧ [Prusti: unsupported feature] unions are not supported [Ln 16, Col 33]
▷ Verify with Prusti   ⓧ Verification of crate 'ex_autorand' failed with 1 error (0.8 s)   rust-analyzer
```

Figure 5: Prusti's output when we verify the code in figure 4 (without any specifications)

.

2.2 Summary

Using the approach described above, we evaluate how well Prusti handles the verification of `unsafe` sections in Rust code. Using Prusti on real-world code examples will show us the capabilities and limitations of Prusti at the moment and will hopefully give us an idea of what features could be added in order to make Prusti more powerful in the future.

3 Core Goals

(1) Analyse a subset of the bugs found by RUDRA and their corresponding fix.

(2) Categorise these examples according to the following criteria:

- What *Rust* features do they use? (e.g. traits, lifetime conversions, casting, transmuting)
- Does Prusti support these features?

(3) Extract the relevant parts of the code and isolate them into self-contained pieces of code. If the code contains features which are known beforehand to not be supported by Prusti, try to rewrite the code without these features – without changing anything relevant to the bug and its fix.

(4) Write specifications and run Prusti on the fixed piece of code.

(5) In the cases where Prusti is unable to verify the fixed code, assess whether this is due to

- Prusti still missing a feature which would allow it to verify this example
- another bug in the code which had not been detected yet
- some other reason

(6) In the cases where Prusti has successfully verified the fixed version of the code, try the verification on the buggy version with the *same specifications.* Prusti should fail to prove this version due to the bug contained in this code!

- If Prusti *can* verify this code successfully, analyse whether
  - our Prusti specifications are buggy.
    In this case, go back to core goal 4 and correct the specifications used for the verification, then continue from there.
  - there is a bug in Prusti.

4 Extension Goals

(1) Pick one of the missing features identified in core goal (5) and add it to Prusti.

   (1) Implement the feature.

   (2) Check whether our adjustment to Prusti was successful, i.e. whether Prusti is now able to correctly handle the targeted examples.

(2) Extend the subset of examples that we selected to analyse and verify in the core goals.

(3) Give instructions on how to write appropriate Prusti specifications to verify unsafe Rust code (i.e. instructions on how to do what we did in the core goals).

5 Schedule

Core Goals	8 weeks
Extension Goals	8 weeks
Writing Report	4 weeks
Reserve	2 weeks

References

[1] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "RUDRA: Finding Memory Safety Bugs in Rust at the Ecosystem Scale," in *SOSP '21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.* Association for Computing Machinery, 2021.

[2] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust Types for Modular Specification and Verification," in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 3, no. OOPSLA. ACM, 2019, pp. 147:1–147:30. [Online]. Available: http://doi.acm.org/10.1145/3360573

[3] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A Verification Infrastructure for Permission-Based Reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62. [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5_2

[4] Prusti Project, Chair for Programming Methodology, ETH Zürich, "Prusti User Guide: Verification Features," accessed: 2023-03-06. [Online]. Available: https://viperproject.github.io/prusti-dev/user-guide/verify/summary.html

[5] Giacomo Stevanato, "String::retain allows safely creating invalid (non-utf8) strings when abusing panic #78498," accessed 2023-04-07. [Online]. Available: https://github.com/rust-lang/rust/issues/78498

[6] ——, "Prevent String::retain from creating non-utf8 strings when abusing panic #78499," accessed 2023-04-07. [Online]. Available: https://github.com/rust-lang/rust/pull/78499

[7] "CVE-2020-36317," accessed: 2023-04-07. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2020-36317

[8] Mike Lubinets, "impl Random on arrays, excerpt from autorand-rs/src/lib.rs," accessed: 2023-03-22, GitHub issue referencing the excerpt: https://github.com/mersinvald/autorand-rs/issues/5. [Online]. Available: https://github.com/mersinvald/autorand-rs/blob/c838309507f9364ecf61553a6ae113dd720fdab9/src/lib.rs#L160-L170

[9] ——, "Fix of RUSTSEC-2020-0103 bug in autorand," accessed: 2023-03-22. [Online]. Available: https://github.com/mersinvald/autorand-rs/commit/565d508993936821950009ec4c7c1e33301db81e

[10] "RUSTSEC-2020-0103 autorand: impl Random on arrays can lead to dropping uninitialized memory," accessed: 2023-03-22. [Online]. Available: https://rustsec.org/advisories/RUSTSEC-2021-0012.html