Panic Freedom in Rust and Modular Verification with Prusti

Niina Okunishi

April 2024

1 Preface

This report is written as a final wrap-up for my project conducted for CS Directed Studies taken in Winter Session 2024. This project aimed to empirically explore how much of Rust's code guarantees panic freedom. I would like to express my gratitude to Dr.Summers, who generously supervised me for this study and on another Rust verification project in the preceding months of summer. I also appreciate the support and suggestions made by the members of the Prusti project team at the Programming Methodology Group of ETH Zürich. I am truly grateful that I had the opportunity to explore the domains of software verification, which is critical for the enhancement of software security.

2 Introduction

One may ask, what is the difference between "proving" program correctness and testing? Similarly, if we could find bugs by testing, what purpose does software verification serve?

The biggest difference lies in the rigour of correctness. Although effective testing strategies can reveal bugs, it is infeasible, and often impossible, to test every possible functionality for every possible input. Formal methods have an advantage on this aspect, since it constructs proofs for all inputs that satisfy the pre-conditions. Theoretically, this omits the possibility that a program misbehaves for some untested cases. Indeed, the defective rate of Central Control Function, an UK air-traffic-management system, was reduced by a factor of 2 to 10 times when formal methods was applied [Hal96]. For this reason, formal methods is most often used in software systems that require high level of security, such as airborne systems [CM14].

In this study, we used Prusti to examine the safety of Rust code, notably, how much of Rust code is guaranteed panic freedom. Panic statements are found in the MIR for functions that have expressions that could hypothetically panic. In other words, the MIR would suggest that a panic could happen, even if that particular expression is panic-free. For instance, we, as programmers, would know that $8 + 9$ would not cause overflows on either 32-bit or 64-bit systems. However, since addition is an operation that could potentially cause overflows for some integers, a panic statement would be found in the MIR. To account for this, Prusti was used to examine how many of the functions are panic-free in practice.

These are important questions to answer since panics are runtime errors; without proper error handling, panicky programs that go unnoticed during production would cause the entire system to crash. Although Rust is considered to have higher memory safety than other languages, restricting every possible error is neither practical nor desirable. Thus, it is the responsibility of each Rust developer to ensure that their programs do not invoke panic.

3 Background

Rust is a low-level language that is syntactically similar to C++, known for its strict type system that enhances memory safety. One distinguishable feature is its restrictive ownership model. By disallowing shared mutable references, the compiler prevents some erroneous behaviours that are common in other imperative languages, such as race conditions and dangling pointers. Together with Prusti, a Rust verification tool that supports modular specifications, users can prove a wide range of program correctness, from crash consistency to panic freedom.

Panic freedom refers to the absence of run-time errors that are unrecoverable yet reachable according to their specifications [Ast+22]. Specifications are the definitions of programs, which often consist of pre-conditions, post-conditions and invariants. They are essential to proving the correctness of programs when using modular verification tools such as Prusti. These tools consume pre-conditions, the state of the program that is assumed to hold before the execution of a function, to verify the post-conditions, which often regard properties of the returned objects or values in memory. Some scenarios in which panic-freedom may not be ensured in Rust are arithmetic overflows, invocations of panicky expressions such as `panic!()`, and out-of-bound array access [Ast+22].

Automated verification tools have close ties with formal methods. For those that are unfamiliar, formal methods use mathematical models to logically prove the state of some program [Woo+09]. Automated verification tools such as Prusti aim to make formal verification more accessible. It is known that formal methods require extensive knowledge of logic such as separation logic and dynamic frames [Ast+22]. This is inevitably one reason that prevents prevalent usage in industrial settings, as not all programmers have suitable training or background experience [CW96]. With automated verification tools, on the other hand, users can prove various properties of their programs while only utilizing first-order logic; this makes verification much simpler and more approachable for a wider audience.

4 Prusti

One advantage of using Prusti is the reduced overhead of annotations. Unlike other proof assistant tools such as Coq, users only need to learn a handful of syntax that is unique to the tool as it supports a wide range of Rust expressions. Furthermore, Prusti automatically deduces any properties that are ensured by the Rust compiler, for instance, the absence of mutable shared references [Ast+22]. In other words, it allows users to focus on properties that are unique to the function, which are perhaps more interesting. This also enhances accessibility, since it allows programmers with less knowledge of memory safety and ownership in Rust to nonetheless verify essential properties of their programs.

4.1 Preconditions with Prusti

One way to prove panic freedom with Prusti is by providing pre-conditions. For instance, the Rust function below can hypothetically cause overflows, namely, if we provide `u32::MAX` − 1 as an argument.

```
fn add(i: i32) {
    let _ = i + 1;
}
```

Figure 1: Function with Potential Overflow

For such functions, we can expect to find a panic statement in the MIR ("Mid-level Intermediate Representation"), which is a control flow produced by the Rust compiler. This implies that the Rust compiler deduces the function to potentially cause panic.

The example above illustrates how specifications are critical when proving panic freedom with Prusti. Without any pre-conditions, verifiers cannot make any assumptions regarding the state of the program. Since the function signature does not guarantee the validity of input values, Prusti will raise a verification error that indicates potential overflows.

Returning to our example, if we know the range of values the input `i` would take on, we could provide a precondition that specifies its range.

```
#[requires(i < i32::MAX - 1)]
```

With this pre-condition, Prusti can deduce that overflows cannot happen with any call to `add(..)`, since the potentially erroneous line, `i+1`, is ensured to produce a result less than the maximum signed 32-bit integer in Rust.

It is worth mentioning that pre-conditions are not just helpful in proofs, but also, they help prevent human errors. For instance, programmers may forget

an invariant they implicitly imposed on a function, and impetuously make a call with an invalid input. On other occasions, they may not handle errors properly. For instance, they may call `Option::unwrap()` without checking if the object it calls on is an instance of `Some(..)`, not `None`. We note that `Option::unwrap()` is a function that attempts to extract the value wrapped by `Some(..)`; thus, calling on a `None` object would cause the program to panic. In the latter example, we could provide a pre-condition that specifies the input is an instance of `Some(..)`. This prevents the panic from happening in runtime, since Prusti will raise a verification error that the pre-condition of the callee function is not satisfied.

4.2 Invariants with Prusti

Invariants are properties that are assumed to remain unchanged throughout the program execution. One type of invariant that is available with Prusti is the loop invariant [1]. Loop invariants are similar to inductive proofs, since Prusti inductively verifies that the given condition is met at the start of each iteration [Ast+22]. They are also similar to pre-conditions given to recursive calls. In the example below, the loop invariant signified by `body_invariant!` must be present to ensure that the increment of `j` does not cause overflows. On the other hand, the increment of `i` does not need the loop invariant, presumably due to the continuation condition of the while loop.

```
fn some_loop(mut i: i32) {
    let mut j = 0;
    while i < 10 {
        body_invariant!(j < i);
        i += 2;
        j += 1;
    }
}
```

Figure 2: Caption

4.3 Post-condition with Prusti

Unlike pre-conditions and invariants, which are assumptions that verifiers take as granted, post-conditions are statements that we want to prove. In Prusti, users can provide post-conditions by the `[#ensures(..)]` tag and refer to the returned object by the `result` keyword.

Given sufficient pre-conditions and invariants, verification tools can prove (or disprove) that functions exhibit intended behaviour. The amount of overhead needed to verify a post-condition varies among functions and conditions. Some

[1] Prusti with versions 2023-08-22-1715 and older do not support for-loops, whereas while loops are supported

post-conditions are verifiable without providing any assumptions, whereas others require the post-condition of the callee function. Assuming `Option::is_some()` is a pure function [2] , the program below requires no pre-condition to verify, since Prusti can statically analyze that the returned value is always `Some(..)`. [3]

```
#[ensures(result.is_some())]
fn return_some() -> Option<u32> {
    Some(10)
}
```

Figure 3: post-condition requires no additional overhead

On the other hand, the function below has a return type that depends on the conditional. Logically, we know that the given post-condition will not always hold.

```
#[ensures(result.is_some())]
fn add(x:u32) -> u32 {
    if x < 10 {
        Some(x)
    } else {
        None
    }
}
```

Figure 4: an example in which a post-condition requires addtional overhead

In this example, Prusti will raise a verification error which conveys that the post-condition may fail to meet. One solution is to impose a restriction on the inputs such that `add(..)` can only be called with arguments less than 10.

```
    #[requires(x < 10)]
```

A more flexible approach is to use implications in the post-condition. This allows users to verify the condition for a wider range of inputs

```
    #[ensures(x < 10 ==> result.is_some())
              && x >= 10 ==> result.is_none()]
```

[2] to call a function from specifications, functions must be marked as pure. Prusti checks if the function is deterministic and side effectful when marked as pure, but it is the responsibility of the users to ensure termination

[3] in Rust, lines without a trailing semi-colon is an implicit return statement

The example above highlights that not all verification errors are caused by post-conditions that are impossible to satisfy, instead, it could be due to insufficient assumptions provided to the tool.

5 Mid-level Intermediate Representation

The Mid-level Intermediate Representation, or the "MIR", is the control flow of the programs produced by the Rust compiler when they are built. They comprise variable declarations and multiple basic blocks, which group a series of statements. The control flow between different blocks is defined by the *terminators* [4], which differs from statements as they could have multiple successors. By default, statements only have one successor, namely, the statement or the terminator that follows.

The MIR helps determine if there is a source of panic within the program. Some examples of MIR and various panic statements can be found here.

5.1 MIR with No Panic Statements

By statically analyzing `panic_free(..)`, we know that this function contains no sources of panic or errors.

```
fn panic_free() {
    let x = Option::Some(10);
    x.unwrap();
}
```

Figure 5: Program with no source of panics

The MIR produced for this function can be seen in Figure 6.

[4] terminators corresponds to the last line of each block, which often contains a right arrow

```
fn panic_free() -> () {
    let mut _0: ();
    let _1: std::option::Option<i32>;
    let _2: i32;
    let mut _3: std::option::Option<i32>;
    scope 1 {
        debug x => _1;
    }

    bb0: {
        _1 = std::option::Option::<i32>::Some(const 10_i32);
        FakeRead(ForLet(None), _1);
        _3 = _1;
        _2 = std::option::Option::<i32>::unwrap(move _3) -> [return:
            bb1, unwind: bb2];
    }

    bb1: {
        _0 = const ();
        return;
    }

    bb2 (cleanup): {
        resume;
    }
}
```

Figure 6: Example MIR for a panic-free program

In this example, the terminator of `bb0` suggests that the flow of the program is conditional after calling `Option::unwrap()`. If `unwrap()` was successful, then the state of the program will move to `bb1`. On the other hand, if an error occurred during this call, then the compiler would call the destructor to unwind the process; this will move the state to `bb2` instead.

Notice how we did not use the word "panic" to describe the second scenario. From the user's perspective, calling `Option::unwrap()` on a `None` object would cause panics at runtime. However, from the compiler's perspective, it cannot deduce that the implementation of `Option::unwrap()` has sources of panic, just from modularly analyzing the function call.

```
fn nest_panic() {
   let x: Option<u32> = Option::None;
   x.unwrap();
}
```

Figure 7: Program that would panic at runtime

Thus, the control flow graph for `nest_panic()` would look very similar to Figure 6. [5] Indeed, the Rust compiler does not raise any compilation errors or warnings for `nest_panic()`, even though it invokes a panic at runtime.

5.2 MIR with Panic Statements

Various sources of panic produce various panic statements. To the best of our knowledge, the panic statements are signified in three ways, in order of most common to least:

1. `assert` statement, with reason of potential panic.
2. A function from the `core::panicking` module (i.e., `panic(..)`). This indicates that a call to `panicking::panic(..)` was made implicitly from within a program, which is a built-in function that declares the panic.
3. `std::rt::begin_panic::<&str>(const "explicit panic")`, which is the case when the `panic!()` marco is called. Again, this implies that the function `std::rt::begin_panic(..)` was implicitly called from the program, which is a function that invokes the panic.

Consider the following function.

```
fn add_overflow() {
   let _ = 1 + i32::MAX;
}
```

Figure 8: Addition Overflow

By definition, we can deduce that an overflow would happen, since the sum of 1 and the maximum signed integer results in a negative integer. However, without any knowledge of overflows, one can also deduce from the MIR that this function contains some expression that could cause addition overflows.

```
assert(!move (_6.1: bool),
    "attempt to compute `{} + {}`, which would overflow",
    move _4, move _5)
-> [success: bb1, unwind: bb2];
```

Figure 9: Panic Statement for Addition Overflow

Note that panic statements in the MIR do not imply that the function would always panic. In the example below, `add_safe()` computes the sum of 10 and

[5] The actual MIR for this example can be found here

20. Since both integers are sufficiently small, their sum is ensured to be smaller than `i32::MAX`.

```
fn add_safe() {
   let _ = 10 + 20;
}
```

Figure 10: No Addition Overflow

However, if we take a look at its MIR, we see that the same `assert` statement appears as the one shown in Figure 9. This is consistent with how `assert` statements are part of terminators. Since MIRs are merely graphs of control flow, rather than indicating if the panic will definitely happen, they suggest all possible flows given an expression. In this example, if the assertion evaluates to true (note the negated boolean), then the operation did not cause overflow, thus, the MIR indicates that the program state will go to `bb1`. Otherwise, an overflow happens, thus, the program will go to a cleanup state in `bb2`.

5.3 Panic Statements for Division and Modulo Operations

Although in most cases, there is at most one panic statement for each expression, a notable case that may produce two assertions are division and modular arithmetic. For both operations, the compiler always checks if the divisor is zero.

```
assert(!move _2, "attempt to divide ‘{}‘ by zero, const 8_i32)
       -> [success: bb1, unwind: bb3]
```

Figure 11: Panic Statement found for Division

```
assert(!move _6, "attempt to calculate the remainder of ‘{}‘ with a
    divisor of zero", _4)
       -> [success: bb1, unwind: bb2];
```

Figure 12: Panic Statement found for Modulo Operation

In addition, if the particular expression involves signed integers, then the compiler also checks for overflows. For this reason, if both divisor and dividend are statically known, then it produces two panic statements; similar to other languages, integers are signed in Rust by default.

6 Problem Statement

This project had an engineering focus rather than theoretical. The database framework was built for the purpose of investigating the following questions.

RQ1: Safety of Rust Code

We aim to explore how much of the Rust code ensures panic freedom by utilizing Prusti and the Rust compiler. In particular, what percentage of Rust functions are panicky in practice? In our analysis, we determined the lower bound on the percentage of functions in which Prusti deduced panic freedom.

RQ2: Deductive ability of Prusti and its Overhead

Our project also aims to contribute to the improvement of Prusti. Notably, we chose two sources of panic, in which Prusti can deduce panic-freedom in some occasion but not all. Through the examples, we explored the scenarios in which Prusti deduces panic freedom without any specifications. For the cases it fails to do so, we determined the minimum overhead needed to verify the post-conditions.

We note that since Prusti is currently in development, the limitations of the tool need to be taken into consideration when conducting data extraction and analysis. Thus, along with the two research questions, the framework also supports the investigation of the current state of the tool.

7 Prototype

The source code can be found here. The framework comprises two major components: running Prusti and obtaining the MIR from rustc, which aligns with our research focus.

7.1 Acknowledgement of Previous Work

In this project, we use Prusti and rustc (the Rust compiler) developed or configured by previous works. More information can be found at `https://www.pm.inf.ethz.ch/research/prusti.html` for Prusti and `https://github.com/willcrichton/flowistry` for Flowistry. This framework assumes that the user has Rust installed; if not, one can do so from this link.

7.1.1 Prusti

This project used the latest version of Prusti (Version: 2023-08-22-1715) which can be download from here. As this project was conducted under a Mac OS environment, the script spans a new command that invokes the x86 version of Prusti, if located in the `/target/debug` directory. Currently, this part of the program is hardcoded and the user has to manually download their version of Prusti from the website.

7.1.2 Flowistry

Flowistry is a static analysis tool that modularly examines the information flow of Rust programs [Cri+22]. We used the rustc configured by the Flowistry project team since it produced significantly less compilation errors compared to the default Rust compiler. Furthermore, the compilation error does not necessarily hinder the extraction of MIR. With the default compiler, a compilation error at some point in the file prevented the MIR from getting produced for any function declared in the same file.

7.2 General Structure

All Rust and Python programs are under the `prusti-eval` directory. When running commands, it automatically creates a `../workspace` directory, in which all gathered data gets dumped into. This directory has three main sub-directories, each of which corresponds to the scripts under the root directory with the same name. Most automation and parsing scripts are written in Python, whereas programs that interact with Prusti or the MIR are written in Rust.

8 Data Collection

We extracted the MIR and the verification results of Prusti from 209 crates downloaded from crate.io, which is a registry for all publicly available Rust APIs. The 209 crates in the `CrateList.json` were originally sampled from CrateList-1000.json, which lists the 1000 most downloaded crates as of 2020-01-14 [Ast+20]. We chose to sample code from crates.io for two reasons. First, source codes published on crates.io are public APIs. This implies that they are more likely to have better error handling and less prone to trivial bugs than codes written for personal purposes. Thus, sampling on such codes allows us to omit panics that happen due to poorly written code. Second, by sampling on APIs, we can conduct our analysis on code that is used by a greater number of programmers.

We acknowledge that the most downloaded crates as of today may differ from three years ago. However, we argue that the order of all-time downloads is unlikely to change drastically in three years; this could be validated by manual inspection on crates.io.

8.1 Running Prusti

The `./prusti` directory stores all code related to running Prusti and extracting the verification results. The script is programmed to dump its results under the `../workspace/prusti` directory. In particular, there are three sub-directories

1. `./crates` : Contains the summary of verification results for each crate.
2. `./crash` : For each crate, error messages produced by Prusti when it internally crashed is dumped here. By running command `crash.py`, one could also obtain a summary of the crashes that happened.
3. `./results`: Contains the summarized results of running Prusti on each crate. To ease the navigation of the files, Rust warnings and information regarding features unsupported by Prusti are dumped separately into `/rust` and `/unsupported` directories. The aggregated result is summarized into a single .json file under `/summary`.

The raw verification results stored in a .txt file can be found under `./data/archive`.

8.1.1 Optimization

We note that running Prusti on large crates could take up to several hours, depending on its size.

Furthermore, to obtain the number of functions Prusti ran on, we currently dump all viper information into `/tmp/log` file, and for each crate, we delete after we extracted its summary. Since the viper information dumped by Prusti

contains an enormous amount of data, the deleting process could take up to 30 minutes, which is the time limit we set to avoid delaying the verification of other crates. Since we only access one of its directories out of many, it may be worth configuring Prusti to dump that directory only.

8.2 Obtaining the MIR

The `./mir` directory contains all program logic related to obtaining the MIR. In particular, one could access the programatic representation of HIR ("High-Level Intermediate Representation") in the src file under the `mir-rust` sub-directory. This is useful when obtaining additional information about each function, such as parameter and `Ty`, the internal representation of Types in the Rust compiler.

The Rust code was written by modifying the Flowistry program to extract the MIR for every function in the file. Since the MIR is the control flow graph of a function, if successful, one can expect to obtain a MIR for each function. For storage optimization, the string representation of the MIR and the path to the function gets dumped within the crate directory downloaded in the `/tmp` directory. It then automatically crates an intermediate summary into the `/tmp/mir` folder.

The `../workspace/mir` directory has three sub-directories:

1. `report`: stores the summary of information obtained from MIR for each crate.

2. `rerun`: stores the new summary of MIR if the script was run twice for a particular crate. This aims to increase the number of functions MIR could be produced by reducing compilation errors. The script attempts to do so by incorporating the suggestions made by the compiler (i.e. syntax, import statements) when it was initially ran. To speed up process, this option could be disabled.

3. `summary`: contains a .json file that summarizes the MIR data of all crates.

8.3 Comparing Results

The `./eval` directory contains programs that compares the results obtained from the MIR and Prusti. The summary for each crate can be found under the `./eval/crates` directory.

To compare if Prusti gave a verification error that corresponds to panic statement in the MIR, we used the `rustc_hir` library to obtain the path to the function. In the case that the compiler produced imprecise or inaccurate end line number (notably, the start and end number of the function are equal), we conducted string search on the file to obtain the actual end number of the function. We then compared the line number that Prusti gave a verification error at, and compared if is within the range of the start and end number. If this condition is satisfied and if the panic statement in MIR and the verification

error was caused by the same reason, then we deemed that Prusti and the compiler both deduced that the function could panic.

We observed that a number of functions had multiple expressions with the same sources of panic; for instance, `update_buffer()` implemented for adler32-1.0.4 had 11 panic statements that all regards addition overflow. In our analysis, we extracted one unique source of panic for each function. In particular, we are interested in analyzing the safety of Rust code per function, rather than per expression, as we sampled Rust APIs. From the users perspective, it is less critical which particular expression in the implementation caused the panic due to encapsulation.

The results were then catergorized into four categories, in the following order.

Case 1

Both the Rust compiler and Prusti deduced that the function may panic for the same reason. In other words, for a panic statement found in the MIR, Prusti raised a verification error that the specific panic may happen. We extracted functions such that the reason and location of the panic were the same.

Case 2

There was a panic statement in the MIR, however, the line had a feature(s) that are unsupported by Prusti. One notable example is bit shifting. Although the Prusti server suggests using the configuration flag `ENCODE_BITVECTORS` to resolve this issue, this flag is noted as highly experimental in the Prusti developers guide [6]. It was indeed tested that this flag causes internal errors, thus, we did not set this flag when running our experiment.

Case 3

The panic statement was only found in the MIR because there a verification or internal error caused by a preceding expression. This halts the verification for that function, thus, Prusti did not evaluate that expression.

Case 4

The panic statement was only found in the MIR but no preceding error was found. This implies that the function had a source of potential panic, but Prusti deduced panic freedom.

The order of this extraction ensures that any function whose source of panic categorized into Case 4 is essentially panic-free. This is due to our per-function analysis. Our script only deems a source to be in Case 4, if the function contained no source of verification error (Case 1) nor unsupported features (Case

[6] `https://viperproject.github.io/prusti-dev/dev-guide/config/flags.html`

2). If either condition fails to be satisfied at any point in the function, then it is categorized into Case 3 instead.

To obtain the aggregate result of all crates instead, run `summary.py`, which produces an overall summary of all crates. The .json files provide an example function with its path for each source of panic, which is useful for manual inspection. For the reason described in the following section, we omitted crates that caused Prusti to crash in this aggregation step. However, the summary of the MIR for the crate can still be found under `./eval/crates`.

9 Limitations and Change of Plans

9.1 Hardware Challenges

It was initially proposed to use *Qrates*, a framework that runs queries on a large number of Rust source codes [Ast+20]. One issue with this approach was that it depends on Rustwide API, which is unsupported on Mac OS. An attempt was made to run on remote Linux servers and virtual machines. However, regardless of the dataset size (i.e., the number of crates of interest), *Qrates* itself requires a large amount of RAM to run queries. This often caused local laptops to crash or exceed the disk storage allocated on remote services. Combined with no apparent alternative to Rustwide and unfamiliarity with Datalog, the focus of the project was shifted from data extraction to framework engineering.

9.2 Prusti Configurations

Prusti halts its verification process for a function after it reaches the first expression that causes some source of error. It was confirmed by the development team that Prusti currently cannot be configured in a way such that it continues verifying after the first encounter. Thus, if a panicky expression was preceded by an expression that halts Prusti, we would not know if Prusti would have raised an error for that expression. This has a significant impact on our data extraction, since there are a large number of Rust features that are commonly used by Rust developers, yet unsupported by Prusti. Since unsupported features are deemed as internal errors, thus, Prusti will stop its evaluation at that line.

Another problem is that Prusti occasionally crashes while running its evaluations. Although Prusti produces valuable verification results before the crash, we omitted these crates since it makes it hard to reason whether Prusti evaluated the expression that had a panic statement in the MIR. There was no particular order in which Prusti evaluates the files within the crate, thus, determining which file and functions it evaluated before the crash requires thorough investigation.

9.3 Rustc Configurations

Despite the improved performance by using the Rust compiler from the Flowistry project, it is still the case that a number of compilation error impedes the MIR from getting produced for that erroneous function. In such cases, it was random whether no control flow graph was produced or if its MIR merely contained the following basic block:

```
bb0 {
    unreachable;
}
```

Figure 13: MIR that signifies compilation error

It should be noted that `unreachable;` statement itself does not indicate a compilation error. Indeed, there were over 1500 valid MIRs that contained the `unreachable` statement in some basic blocks.

9.4 String Representation of Prusti and MIR

In our extraction, we parsed through the text files in which we redirected the stdout to when we ran Prusti. Similarly, panic statements were extracted from the MIR from the text file it was dumped onto. One limitation of this approach is that it makes it hard to reason which panic statement in the MIR corresponds to which verification error. In particular, since the MIR file is produced for each function, it does not indicate which line within that function produced that error. Thus, in our analysis, we made deductions based on reasoning of panic and line numbers.

One scenario in which our approach does not work effectively is when there are multiple sources of panic within the same function. To elaborate, consider the case that Prusti does not give internal errors throughout the function and it gives a verification error for some succeeding expression. Theoretically, for the sources of panic that come before that expression, Prusti does not give any errors because it deduced not to panic. However, our script does not distinguish whether no verification errors were raised because Prusti did not evaluate that line or if it was because it deduced not to panic. Thus, our framework can be improved by interacting with a programmatic representation of Prusti instead, such that we have access to the MIR that its verification results were based on. Another approach is to use a programmatic MIR that has access to the specific line number that produced the panic statement.

10 Result

We ran Prusti and obtained the MIR for 209 crates, all of which were downloaded from crates.io. In our analysis, we only considered the 175 crates in which Prusti did not internally crash during the verification process. For those that did, there were 9 distinct reasons for crashes, in which the most common reason was `Option::unwrap()` called on a `None` value. Note that this error happened internally within Prusti, and the crash was not caused because the runtime error was observed for some function within a crate that Prusti was evaluating.

Panic Statements in the MIR

Since we base our analysis on the control flow graph that the compiler produces, we only considered the functions such that it produced some "valid" MIR. This excludes the ones that suggest compilation errors (Section 9.3), which we refer to as "invalid" MIR in this report. Also, our script produces MIR files that are essentially duplicates for functions implemented in imported modules. Thus, the total number of functions is defined to be the number of unique, valid MIRs that the compiler produced.

Our results show that out of 13903 functions, 3921 of them produced a valid MIR. The 9982 functions that had invalid MIR produced a total of 70785 compilation errors. The three most common sources of compilation errors are summarized below.

Reason	**Total**	**Crate**	**Message**
Import	67112	aho-corasick-0.7.6	"unresolved import ‘automaton‘"
Rust	662	foreign-types-0.5.0	"internal compiler error, compiler/rustc_hir_typeck/src/lib.r ... can't type-check body of DefId(0, 28 č hunked_encoder)"
Syntax	321	tokio-0.2.9	"expected identifier, found keyword ‘let‘"

Figure 14: Top 3 reasons of Compilation Errors

We note that errors categorized under "import" can vary from unresolved imports of crates to global variables. Our script incorporates the suggestions made by the Rust compiler that aims to resolve such issues when the rerunning option is enabled. This increases the number of MIRs that get produced, however, the difference was marginal.

Out of the 3921 functions that panicked, 222 functions had one source of panic, whereas 236 functions had multiple sources of panics; thus a total of 458 functions had some source of panic in their program. In total, 1786 panic statements and 22 distinct reasons of panics were observed.

Total	Source	Crate-Version	File, Function
441	"attempt to compute '{} + {}', which would overflow"	adler31-1.0.4	src/lib, remove
373	"index out of bounds: the length is but the index is "	crc32fast-1.2.0	src/combine, gf2_matrix_square
207	"attempt to compute '{} - {}', which would overflow"	slab-0.4.2	src/lib, remove
154	Assertion Failed i.e. data.len() ≤ CC_LONG::max_value() as usize	native-tls-0.2.3	src/imp/security _framework, has
119	"attempt to divide '{}' by zero"	crossbeam-channel - 0.4.0	examples/ stopwatch, show
111	"attempt to calculate the remainder of '{}' with a divisor of zero"	miniz_oxide-0.3.5	src/inflate/core, validate_zlib_header
105	"attempt to shift left by '{}', which would overflow"	idna-0.2.0	src/uts46 decode_slice
96	"attempt to shift right by '{}', which would overflow"	adler32-1.0.4	src/lib, from_value
80	"attempt to compute '{} * {}', which would overflow"	cc-1.0.50	src/lib, assemble
34	"attempt to compute '{} / {}', which would overflow	nix-0.16.1	src/sys/time, num_microseconds

Figure 15: Top 10 Reasons of Panic

For all panic statements caused by assertion macros such as `assert!(..)` and `assert_eq!(..)`, we counted the total number and grouped them under "Assertion Failed" (row 4 of Figure 15). Thus, the example given in the table is merely one instance of possible assertion failure.

Panic statements for three panicky macros were also observed. In the table below, each row corresponds to data collected for `unreachable!()`, `unimplemented!()`

and `panic!()`, respectively.

Total(Rank)	Reason	Crate-Version
14(11)	"entered unreachable code"	nix-0.16.1
11(13)	"not implemented"	tokio-fs-0.2.0-alpha.6
4(16)	"explicit panic"	tokio-sync-0.2.0-alpha.6

Figure 16: Ranks of Panic Statements Regarding Macros

We note that the number of panic sources do not necessarily correspond to the number of panicky expressions within a function. It was observed that a number of expressions comprised multiple arithmetic operations, which by default would each produce panic statements in the MIR. Furthermore, it is possible that one operator produces two panic statements (Section 5.3).

Comparison with Verification Errors produced by Prusti

Out of the 175 crates, Prusti evaluated 19169 functions and raised a total of 466 verification errors with 15 distinct reasons. Furthermore, 5881 Rust features were reported to be unsupported by Prusti, which consists of 139 distinct reasons. We summarized some data regarding Prusti in the table below.

	Total	Distinct Reason
Verification Errors	466	15
Unsupported Feature	5881	139
Internal Errors	925	n.a
Crashed	34	9

Figure 17: Statistics for Prusti

For internal errors, our script does not keep track of the number of distinct reasons, as there is not much distinction in the error messages.

The five most common sources of verification errors are summarized in the Appendix. Note the error message "value might not fit into the target type." (row 2, Figure 27) was observed for expressions with casting that specifically use the `as` keyword. This error is not relevant to our study, as casting is statically panic free.

Out of the 1786 panic statements, we sampled 320 unique sources of panic per-function basis. For each case defined in 8.3, we summarized our results as follows.

	Total	Most Common Source of Panic (Subtotal)
Case 1	36	”attempt to compute ‘ + ‘, which would overflow” (13)
Case 2	161	”iterators are not fully supported yet” (22)
Case 3	66	n.a.
Case 4	57	”attempt to compute ‘ + ‘, which would overflow” (10)

Figure 18: Caption

Each row in Figure 18 corresponds to `"match"` (Prusti and Rust compiler suggested panic), `"us"` (unsupported features), `"unevaluated"` (Prusti did not evaluate the expression) and `"mir_only"` (Only the Rust compiler suggested panic) entry in the .json file.

The most common source of panic for expressions that were unevaluated by Prusti is omitted from the table, as the reason of panic is not relevant to why it was unevaluated.

11 Discussion

11.1 RQ1: Safety of Rust Code

Our results show that $(458/3921) \times 100 \approx 11.68$ percent of our Rust functions contained some sources of panic. Functions that had multiple and single sources of panic in the MIR account for 6.02 percent and 5.66 percent of all compiled functions, respectively. Since there were approximately the same number of functions that contained single and multiple panic statements, we deduce that the sources of panic are generally distributed among panicky functions. From this, we assumed that the chance of Prusti deducing a panic is roughly equal among all panicky functions, regardless of the number of panic sources it contained.

Our results obtained from Prusti suggest that $(57/320) \times 100 \approx 17.81$ percent of potential sources of panic do not invoke panic. As mentioned in Section 8.3, all functions that had panic statements categorized into Case 4 are guaranteed panic freedom. Also, we note that our analysis was based on a sub-sampled group. Therefore, we deem 17.8 percent to be the lower bound on the percentage of functions that are panic free in practice.

From this, we deduce that $11.68 \times (1 - 0.1781) \approx 9.60$ percent of the functions we considered are not guaranteed panic freedom. We view this percentage to be higher than expected, as it implies that more than 1300 out of the 13903 sampled functions are expected to be panicky. However, it may be possible to drastically improve this number, considering that $(5881 + 925)/19169 \approx 35.5$ percent of errors produced by Prusti were caused by unsupported features and internal errors. Since the verification process halts at the first error, eliminating such errors may improve the number of functions that get proven panic free.

Also, we observed that the number of functions that produced compilation error is high, as it accounts for $(9982/13903) \times 100 \approx 71.8$ percent of sample functions. However, we argue that this is less concerning than it seems, since

1. The 94.8 percent of the errors are produced by unresolved imports of externally defined modules, crates, and variables. Since the Rust compile generally produces the MIR modularly, dependency on outer definitions are known to cause issues. However, this is not an issue when users use the API, as dependencies will be resolved dynamically (assuming it is properly imported).

2. 0.9 percent of Rust errors are either caused by internal compilation errors caused by the interaction with HIR or compatibility issues some versions of Rust. The former is only an issue with this particular experiment and the latter is an unavoidable issue.

Furthermore, it may be possible to reduce this number by improving syntactical issues. One approach that may work is to use a compiler that has higher compatibility with the versions of the crates used in analysis.

11.2 RQ2: Deductive Ability of Prusti and its Overhead

As we observe from our aggregated results, Prusti can deduce that panics do not happen, even if a panic-statement was found in the MIR for that expression. In the following sections, we discuss the possible reasons that Prusti can deduce panic freedom in one case but not in the other, as well as the overhead needed for the failing cases.

11.2.1 Addition Overflow

Theoretically, overflows are panics that only occur for signed integers. For addition, subtraction and multiplication, panic statements for overflows were found regardless of whether signed or unsigned integers were used; this is in contrast with division and modular operations, which was discussed in Section 5.3.

In some cases of addition operations, Prusti can deduce that the panic will not happen. For instance, the function below had a panic statement regarding addition overflow in the MIR. However, Prusti did not raise any verification errors nor internal errors, which implies that it deduced not to panic.

```
fn do1(adler: &mut u32, sum2: &mut u32, buf: &[u8]) {
    *adler += u32::from(buf[0]);
    *sum2 += *adler;
}
```

Figure 19: adler32-1.0.4/src/lib.rs

On the other hand, both the Rust compiler and Prusti suggests that an overflow is possible for the function below, in which the `.pos` field is declared to be an instance of `usize`.

```
fn consume(&mut self, amt: usize) {
       self.pos = cmp::min(self.pos + amt, self.cap);
}
```

Figure 20: flate2-1.0.13/src/bufreader

As we observe, both examples involve the addition of unsigned integers. One hypothesis that the second example produced a verification error is that it references a struct field. In other words, the type of `self.pos` is not explicitly known from within the function declaration. It is unlikely that the type `usize` is causing this error, since changing to `u32`, an integer type whose size is consistent across different targets, produced the same issue.

In fact, in this example, any pre-condition that restricts the sum of `self.pos` and `amt` to be less than the maximum `usize` fixes the error.

```
#[requires(self.pos + amt <= usize::MAX)]
```

Figure 21: An Example of Effective Pre-Condition

11.2.2 Assertions

Another source of panic that Prusti can deduce the absence of panic is assertions. Since all assertions can panic if the given statement evaluates to `False`, we can expect to find a panic statement in the MIR for each asserted expression.

One scenario in which Prusti deduces not to panic is assertions that checks that a value is within the possible range for a given type. Consider the following example, in which `MAX_LENGTH` was defined as `u32::MAX as usize`.

```
fn from(src: &'a [u8]) -> Self {
       assert!(src.len() > 0);
       assert!(src.len() <= MAX_LENGTH);
       ...
}
```

Figure 22: iovec-0.1.4/src/sys/windows

In the example above, both assertions produced panic statements that indicate that the expression may fail to evaluate to True and thereby cause panic. However, Prusti did not produce any verification error for neither assertions. Some possible reasons that Prusti can correctly deduce `src.len()` is within a valid range are

1. `slice::len()` (arrays in Rust are coerced into slices) returns a `usize`. Theoretically, its return value cannot be greater than the maximum possible value for `usize`. Prusti may know the return types for certain built-in functions.
2. `src` is an instance of an array. Arrays in Rust has statically known sizes, and when an array of empty size is created, it value is immediately dropped. Thus, it is feasible to deduce that a valid reference to an array is a reference to an non-empty array.

In contrast, Prusti raised a verification error for the example below.

```
pub(crate) fn is_at_index(&self, index: usize) -> bool {
       debug_assert!(offset(index) == 0);
       self.start_index == index
   }
```

Figure 23: From tokio-sync-0.2.0-alpha.6/src/mpsc/block.rs

The callee function `offset(..)` conducts a bitwise-AND operation with `BLOCK_MASK`, which is declared as the negation of `super::BLOCK_CAP - 1`. Furthermore, `BLOCK_CAP` is defined to be 32 on a 64 bit target and 16 bit target on a 32 bit target. Thus, the `debug_assert!(..)` evaluates to true only if `index` is smaller than 31 on a 64 bit target and 15 on a 32 bit target. In other words, the result of the assertion is conditional on the input value `index`, which is a scenario in which specifications are necessary.

It is also unlikely that the location of the constant declaration for `BLOCKCAP` being in the parent module causes the problem. In a similar example shown below, Prusti can deduce that the asserted expression evaluates to True.

```
const S1:i64 = 100;
fn main() {}

mod h1 {
    use super::S1;

    fn main() {assert!(S1 == 100);}
}
```

Figure 24: An example in which a constant is imported from the parent module

For this example, two steps can be taken to eliminate the verification error. For simplicity, assume a 32-bit environment. The first step is to add a pre-condition that imposes an upper bound to the input.

```
// assuming 32-bit target
#[requires(index <= 31)]
pub fn is_at_index(&self, index:usize) -> bool {
    debug_assert!(offset(index) == 0);
}
```

Figure 25: Pre-Condition that restricts the Range of Inputs

However, this pre-condition itself is insufficient, as the returned values by `offset(..)` are conditional on the input value. Thus, by specifying that this function returns zero for any input less than 31, Prusti can deduce that the assertion will not panic.

```
#[trusted]
#[ensures(index <= 31 ==> result == 0)]
pub fn offset(&self, index:usize) -> bool {
   index & !31
}
```

Figure 26: Post-Condition that specifies the Range of Outputs

This is an example in which the proof requires post-conditions of functions down the call stack. We note that the current version of Prusti does not support bitwise operations on non-boolean types. Thus, we tag `offset(..)` as a trusted function using `#[trusted]` to suppress this error.

12 Conclusion and Future Work

Based on our findings, we deduce that the majority of Rust codes that developers use are panic-free. However, given that 9.60 percent of functions are predicted to contain some sources of panic, software with higher security may need extensive verification to prove the absence of potential panic.

For future works, some improvements that could be made to the current framework are:

1. Optimization for storage in `/tmp/log` folder. Currently, Prusti is not configured such that it only dumps `vir_program_before_foldunfold` when producing viper folders.

2. Optimize running Prusti when running on larger crates. Also, verification on some crates such as *syn-1.0.13* and openssl-0.10. did not seem to make any progress after an hour. It may be worth investigating if the process is silently failing.

Based on the results and analysis we obtained, we could impose another empirical question, namely, can we improve Prusti such that panic-freedom can be automatically proved in cases it currently fails? The value of the question lies in improving the accessibility of program verification. Users will be able to discard properties that they implicitly assume due to type constraints, which allows them to focus on the core properties of their program.

References

[CW96] Edmund M. Clarke and Jeannette M. Wing. "Formal methods: state of the art and future directions". In: *ACM Comput. Surv.* 28.4 (Dec. 1996), pp. 626–643. ISSN: 0360-0300. DOI: 10.1145/242223.242257. URL: https://doi.org/10.1145/242223.242257.

[Hal96] A. Hall. "Using formal methods to develop an ATC information system". In: *IEEE Software* 13.2 (1996), pp. 66–76. DOI: 10.1109/52.506463.

[Woo+09] Jim Woodcock et al. "Formal methods: Practice and experience". In: *ACM Comput. Surv.* 41.4 (Oct. 2009). ISSN: 0360-0300. DOI: 10.1145/1592434.1592436. URL: https://doi.org/10.1145/1592434.1592436.

[CM14] Darren Cofer and Steven Miller. "DO-333 Certification Case Studies". In: *NASA Formal Methods*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Cham: Springer International Publishing, 2014, pp. 1–15. ISBN: 978-3-319-06200-6.

[Ast+20] Vytautas Astrauskas et al. "How Do Programmers Use Unsafe Rust?" In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428204. URL: https://doi.org/10.1145/3428204.

[Ast+22] Vytautas Astrauskas et al. "The Prusti Project: Formal Verification for Rust". In: (2022). Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, pp. 88–108.

[Cri+22] Will Crichton et al. "Modular information flow through ownership". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 1–14.

Appendices

The following Table summarizes the top five verification errors that were produced by Prusti. A full list can be found here

Total	Message	Crate	Expression
67	”panic!(..) statement might be reachable”	iovec-0.1.4	`panic!("..")`
64	”value might not fit into the target type.”	atty-0.2.14	`Some(i) => Some(i as u128)`
48	”assertion might fail with attempt to add with overflow”	crossbeam -epoch -0.8.0	`Self::Output::new(self + other.re, other.im)`
44	”assertion might fail with attempt to subtract with overflow”	bytes-0.5.3	`self.capacity() - self.len()`
30	”the asserted expression might not hold”	rustc -serialize -0.3.24	`debug_assert!(offset(index) == 0)`

Figure 27: Top 5 Verification Errors